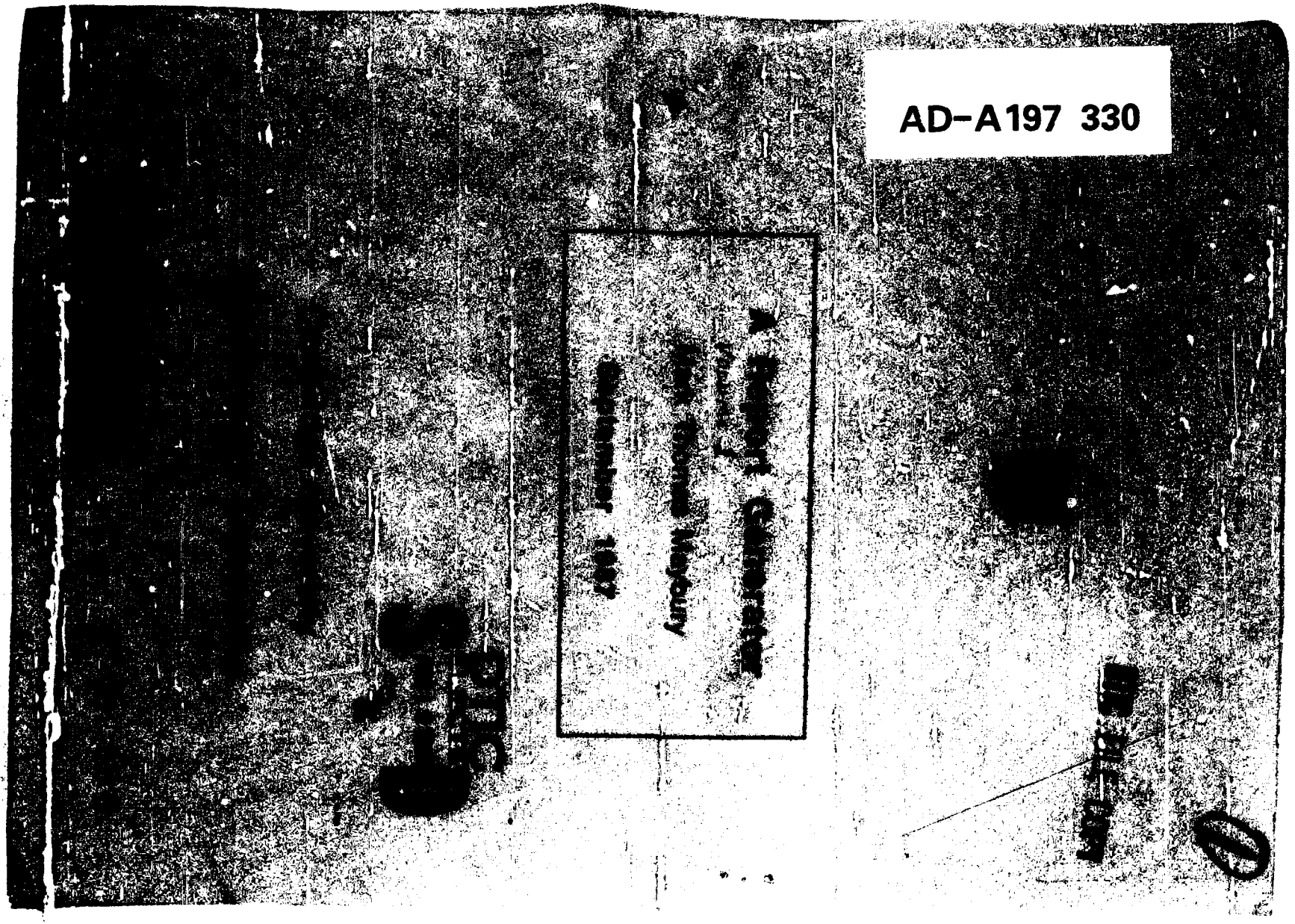


A Report Generator

Volume II

Mark T. Maybury

Cambridge University
Engineering Department

Trumpington Street
Cambridge CB2 1PZ

Wolfson College, 1987

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/CI/NR 88- 139	2. GOVT ACCESSION NO. AD-A202 [illegible]	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A REPORT GENERATOR VOL II - SOURCE CODE		5. TYPE OF REPORT & PERIOD COVERED MS THESIS 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) MARK THOMAS MAYBURY		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS AFIT STUDENT AT: CAMBRIDGE UNIVERSITY, UNITED KINGDOM		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE 1988 13. NUMBER OF PAGES 250
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) AFIT/NR Wright-Patterson AFB OH 45433-6583		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

DISTRIBUTED UNLIMITED: APPROVED FOR PUBLIC RELEASE

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

SAME AS REPORT

18. SUPPLEMENTARY NOTES

Approved for Public Release: IAW AFR 190-1
LYNN E. WOLAVER
Dean for Research and Professional Development
Air Force Institute of Technology
Wright-Patterson AFB OH 45433-6583
21 July 88

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

ATTACHED

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Contents

CHAPTER 1

Introduction

This document is volume II of an M. Phil. thesis representing a text generation system. This portion contains detailed system software listings for the interested implementor. All code is clean and simple to understand. Mnemonic variables and function names pervade the software.

1.1 Software Methodology

The software methodology adopted follows from the top-down refinement approach to structured programming. System modularity together with localisation of procedures combine to provide efficiency and consistency. Furthermore, global variables are minimised, iterative constructs are replaced with elegant tail recursion, and goto statements are forbidden. This good programming practise, consistent throughout the generation system, should make the code very accessible and, hopefully, reusable with minimal effort. Where primitive functions were repeated frequently, macros were developed for rapid execution.

1.2 System Organisation

The software is listed according to the flow of information in the system. First, the main module, which loads in all the sub-modules, is presented. Following this are the mechanisms which interface the linguistical representations with the underlying knowledge representation. Then text strategies are illustrated followed by software for semantic, relational and syntactic analysis. A significant investigation was performed on the development of a successful focus selection algorithm, which is also presented.

The generation module has a feature dictionary system together with unification routines for the comparison of syntactic features and generation of well formed grammatical constituents. Finally, the realisation routines are presented (including linearisation, morphological synthesis and orthographic formating). The knowledge sources for testing are presented followed by some system output.

All software is well documented and includes a title, purpose, copywrite, and often a linguistic theory or principle behind the code. Each function is described in English so that even hacker's who speak dialects other than LISP can understand.

CHAPTER 1

Introduction

This document is volume II of an M. Phil. thesis representing a text generation system. This portion contains detailed system software listings for the interested implementor. All code is clean and simple to understand. Mnemonic variables and function names pervade the software.

1.1 Software Methodology

The software methodology adopted follows from the top-down refinement approach to structured programming. System modularity together with localisation of procedures combine to provide efficiency and consistency. Furthermore, global variables are minimised, iterative constructs are replaced with elegant tail recursion, and goto statements are forbidden. This good programming practise, consistent throughout the generation system, should make the code very accessible and, hopefully, reusable with minimal effort. Where primitive functions were repeated frequently, macros were developed for rapid execution.

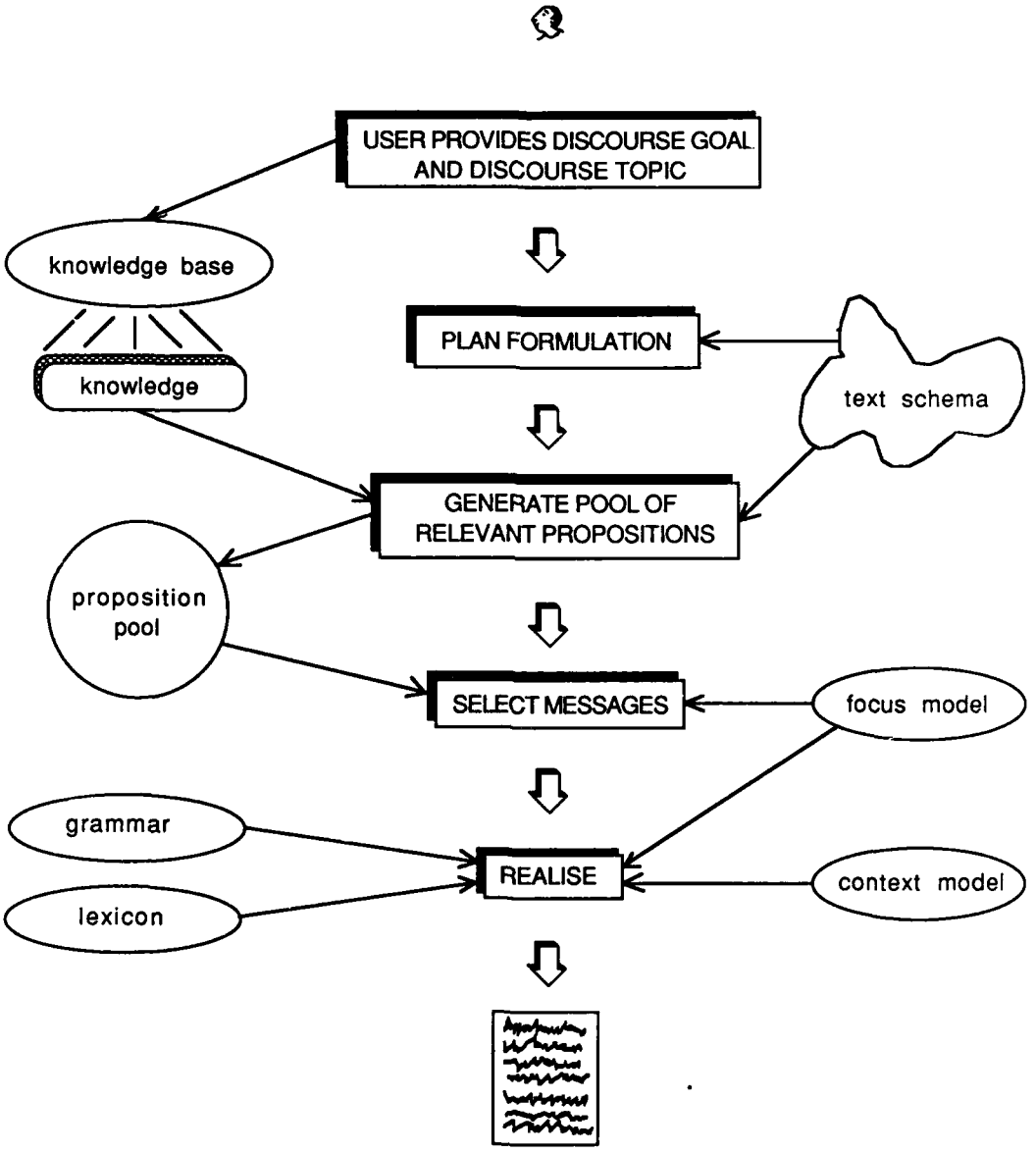
1.2 System Organisation

The software is listed according to the flow of information in the system. First, the main module, which loads in all the sub-modules, is presented. Following this are the mechanisms which interface the linguistical representations with the underlying knowledge representation. Then text strategies are illustrated followed by software for semantic, relational and syntactic analysis. A significant investigation was performed on the development of a successful focus selection algorithm, which is also presented.

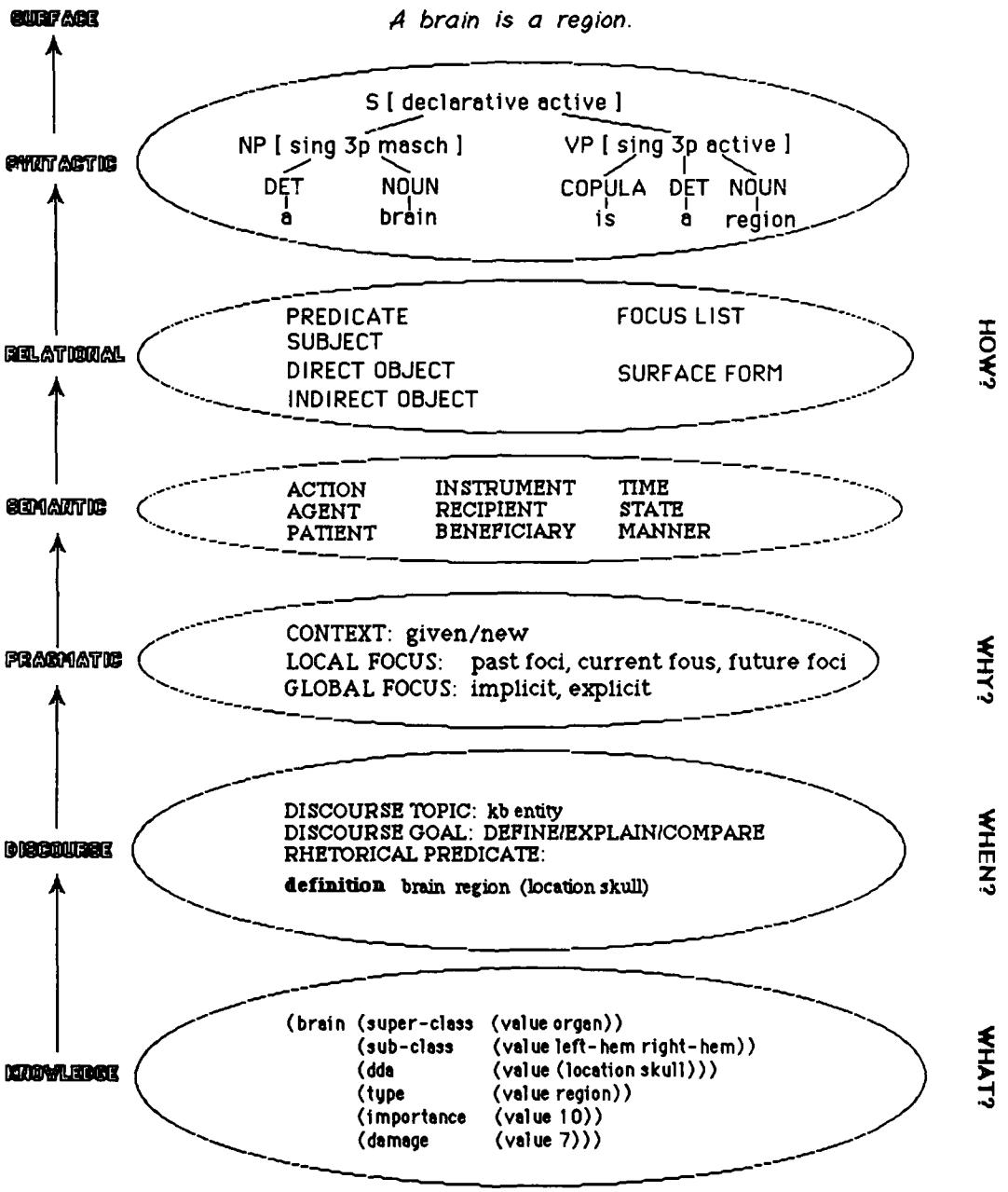
The generation module has a feature dictionary system together with unification routines for the comparison of syntactic features and generation of well formed grammatical constituents. Finally, the realisation routines are presented (including linearisation, morphological synthesis and orthographic formating). The knowledge sources for testing are presented followed by some system output.

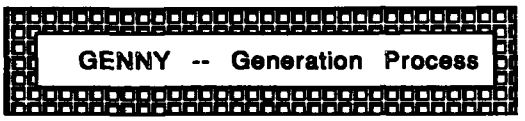
All software is well documented and includes a title, purpose, copywrite, and often a linguistic theory or principle behind the code. Each function is described in English so that even hacker's who speak dialects other than LISP can understand.

GENNY'S KNOWLEDGE AND PROCESSES



FUNCTIONAL LINGUISTIC FRAMEWORK





Select Text Structure

Determine Discourse Goal

Focus Globally on Relevant Knowledge

Create a Knowledge Vista

Instantiate Rhetorical Predicates using KB

Focus Locally on Connected Knowledge

Select Rhetorical Predicates

Transform Rhetorical Proposition to Deep Case Structure

Make Surface Decisions constrained by Pragmatics

Universal Language Representation for Portability

Sentential Realization

lexical translation of knowledge base predicates
unification with grammar
linearization
morphological generation
orthographic synthesis

SECTION 2

MAIN MODULE

```
;////////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                              |
;  MODULE:    TEXT GENERATOR -- MAIN MODULE                                    |
;  PURPOSE:   To generate text from a frame based expert system for            |
;             diagnosis of brain disorders.                                    |
;             The system employs text structures and focus mechanisms          |
;             to synthesise well focused and coherent text.                    |
;             The system can be viewed as consisting of a strategic and        |
;             tactical component.                                              |
;                                                                              |
;  STRATEGIC: KB query --> Text Stuctures --> Rhetorical Predicates            |
;                                                                              |
;  TACTICAL:        Messages                                                   |
;               --> Deep Case                                                  |
;               --> Relational Grammar                                         |
;               --> GPSG Unification Grammar with Features & Function          |
;               --> Linearizer                                                 |
;               --> Morphology and Orthography                                 |
;                                                                              |
; LINGUISTIC                                                                   |
; PRINCIPLES:  Analysis of human produced text suggests that individuals       |
;              utilise common strategies of discourse to achieve a             |
;              particular discourse goal.  They are constrained in realising   |
;              this discourse plan by choosing knowledge salient to the        |
;              discourse topic as well as uttering acts which are connected    |
;              via their focus of attention as well as their role in the       |
;              text (as suggested by content, verb choice, and lexical         |
;              connectives). GENNY addressed three discourse goals:            |
;              DEFINE, EXPLAIN, and COMPARE.  Furthermore, the generator       |
;              suggests a representation which holds promise to be             |
;              independent of text-type, domain, knowledge formalism and       |
;              (possibly) language.                                            |
;                                                                              |
;  OWNER:    copywrite Mark T. Maybury, May, 1987.                             |
;                                                                              |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////////

    (include support.1)                ; system support routines


; STRATEGIC GENERATION
; ~~~~~~~~~~~~~~~~~~~~

  (include focus.1)                    ; focus of attention strategies/plans
  (include anaphora.1)                 ; pronominalization devices
  (include kb_interface.1)             ; interface to kb
  (include predicates.1)               ; rhetorical predicate instantiation
  (include text.1)                     ; theme-scheme text structures

; FUNCTIONAL TRANSLATION
; ~~~~~~~~~~~~~~~~~~~~~~

(include translate.1)                  ; rhetorical predicate -> deep case structure
(include relationalgram.1)             ; deep case structure  -> relational grammar

; TACTICAL GENERATION
; ~~~~~~~~~~~~~~~~~~~

(include generate)                     ; surface tree generator with unification grammar
(include realization.1)                ; realization routines
(include morphsyn.1)                   ; morphological synthesis routines
(include surface_form.1)               ; produce surface form

; LINGUISTIC + DOMAIN KNOWLEDGE
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

(include dictionary.1)                 ; dictionary
(include grammar)                      ; include grammar
(include initialise)                   ; system initialisation

(main)                                 ; call main routine
```

SECTION 3

SUPPORT MODULE

```
;////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                           |
;  MODULE:   INITIALISE                                                     |
;  PURPOSE:  To initialise top level system variables and knowledge         |
;            structures.                                                    |
;  OWNER:    copywrite Mark T. Maybury, July, 1987.                         |
;                                                                           |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////

(preprocess *grammar*)                  ; preprocess grammar for efficiency
(setq *tracing* 1)                      ; set level of tracking
(setq *given* nil)                      ; no discourse history
(setq *predicate-types*                 ; rhetorical predicate types for checking

      '(definition attributive constituent illustration evidence
        example cause-effect compare~contrast inference))
```

support.l Wed Aug 19 00:37:52 1987

```
;////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                           |
;  MODULE:   TEXT GENERATOR -- SUPPORT MODULE                               |
;  PURPOSE:  To provide support routines for the text generation system.     |
;            These include general support routines for macros, io, and     |
;            debugging.                                                     |
;  OWNER:    copywrite Mark T. Maybury, June, 1987.                         |
;                                                                           |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////

    ;*** declare global variables, surrounded by "*" for uniqueness ***

(declare (special *predicate-types* *given*)
         (macros t))

    ;*** general support routines ***

 (include ~/lisp/lispaids/macros.l)   ; useful macro primitives
 (include ~/lisp/lispaids/io.l)       ; useful lisp io module/interface support
 (include ~/lisp/lispaids/stack.l)    ; stack support routines

  ;** following used for debugging:

 (include ~/lisp/semantics/save.l)    ; routine to save items to files
 (include ~/lisp/lispaids/track.l)    ; tracking functions
```

macros.l Wed Aug 19 03:40:00 1987

```
;*****************************************************************************
;                                                                            *
;  FUNCTION: Macro definitions (first, second, third, head, tail, etc)       *
;  PURPOSE:  To perform certain basic operations quickly.                    *
;  INPUT:    Varys, but generally some type of list.                         *
;  OUTPUT:   Some member of the given list or a value such as t or nil.      *
;                                                                            *
;*****************************************************************************


; args returns arguments of a given macro argument binding -- ie it returns
;   the cadr of the given parameter binding.

(defun args macro (lyst) '(cadr ,(cadr lyst)))
(defun fexprarg  macro (lyst) (list 'car (cadr lyst)))

;
;     List accessing functions:
;

(defun first   macro (lyst) '(car ,(args lyst)))
(defun second  macro (lyst) '(args ,(args lyst)))
(defun third   macro (lyst) '(caddr ,(args lyst)))
(defun fourth  macro (lyst) '(cadddr ,(args lyst)))
(defun fifth   macro (lyst) '(caddddr ,(args lyst)))
(defun sixth   macro (lyst) '(cadddddr ,(args lyst)))
(defun seventh macro (lyst) '(caddddddr ,(args lyst)))
(defun eighth  macro (lyst) '(cadddddddr ,(args lyst)))
(defun ninth   macro (lyst) '(caddddddddr ,(args lyst)))
(defun tenth   macro (lyst) '(cadddddddddr ,(args lyst)))


(defun head   macro (lyst) '(car ,(args lyst)))
(defun tail   macro (lyst) '(cdr ,(args lyst)))
```

```
;*****************************************************************************
;                                                                             *
;  MODULE:  IO -- LISP INPUT/OUTPUT TOOLBOX                                   *
;  PURPOSE: To facilitate the development of user interfaces.                 *
;                                                                             *
;*****************************************************************************

;#############################################################################
;#############################################################################
;                                                                             #
;                    TERMINAL PRINTING FUNCTIONS                              #
;                                                                             #
; (print-list list &optional punctuation)                                     #
; (print-sentence sentence &optional punctuation)                             #
; (blank number-of-lines)                                                     #
; (space number-of-spaces)                                                    #
; (tell-user message &optional numblanksbefore numblanksafter spacesbefore)   #
; (print-list list-of-items)                                                  #
; (writeln list-of-strings-atoms-lists-or-numbers)                            #
; (print-list-commas-and list-of-items)                                       #
;                                                                             #
;#############################################################################
;#############################################################################

;*****************************************************************************
;                                                                             *
;  FUNCTION: all-but-last                                                     *
;  PURPOSE:  To return all but the last item in a given list.                 *
;  INPUT:    list                                                             *
;  OUTPUT:   list minus the last element                                      *
;                                                                             *
;*****************************************************************************

(define (all-but-last list)
;==========================
  (cond ((null (tail list)) nil)                ; tail empty --> nil
        (t (append (list (head list))           ; attach head of list to
                   (all-but-last (tail list)))  ; all but last of tail
           )
        )
  )
)

;*****************************************************************************
;                                                                             *
;  FUNCTION: length-characters                                                *
;  PURPOSE:  To return the length of the characters (plus 1 space between)    *
;            in a given list.                                                 *
;                                                                             *
;*****************************************************************************

(define (length-characters lyst)
;###############################
 (cond
  ((null lyst) 0)
  (t (+ (length (explode (head lyst)))          ; length of first
        (length-characters (tail lyst))         ; length of rest
        1                                       ; space between words
     )
  )
 )
)

;*****************************************************************************
;                                                                             *
;  FUNCTION: print-list                                                       *
;  PURPOSE:  To print out a list of atoms with a space between each           *
;            followed by proper punctuation if requested.                     *
;  INPUT:    list of words                                                    *
;  OUTPUT:   Words separated by one space, with no spaces at end.             *
;  AIDED BY: my-patom     == prints the given atom                            *
;            last         == returns the last item in a list.                 *
;                                                                             *
;*****************************************************************************

(define (my-patom atom) (patom atom) (princ " "))
;################################################

(define (print-list lyst &optional punctuation)
;##############################################
 (mapc 'my-patom  (all-but-last lyst))          ; space after all but last
 (patom (car (last lyst)))                      ; print the last one
 (cond                                          ; punctuate properly
   ((eq punctuation 'space) (princ " "))        ;  - put a space at the end
   ((eq punctuation 'period) (princ "."))       ;  - period if selected
   ((eq punctuation '?) (princ "?"))            ;  - question
```

```
    ((eq punctuation '!) (princ "!"))            ;  - exclamation
  )
)

;*****************************************************************************
;                                                                             *
;  FUNCTION: writeln and writeout                                             *
;  PURPOSE:  To output a list of atoms, numbers, lists and strings with       *
;            proper spacing and punctuation, as requested.                    *
;  INPUT:    A list of output and optional punctuation.                       *
;  OUTPUT:   Nice Pascal like output.                                         *
;                                                                             *
;*****************************************************************************

(defun writeln fexpr (outputlist)
;#################################
 (writeout outputlist)                           ; call writeout fexpr
 (terpri)                                        ; blank one line
)

(defun write1 fexpr (outputlist)
;###############################
 (writeout outputlist)                           ; call writeout fexpr
)


(define (writeout outlist)
;##########################
  (mapc 'writeout1 (all-but-last outlist))             ; writeout1 all but last elements
  (cond                                                ; punctuate properly
   ((eq (car (last outlist)) 'space) (princ " "))      ;  - put a space at the end
   ((eq (car (last outlist)) 'period) (princ "."))     ;  - period if selected
   ((eq (car (last outlist)) '?) (princ "?"))          ;  - question
   ((eq (car (last outlist)) '!) (princ "!"))          ;  - exclamation
   (t (writeout1 (car (last outlist))))                ;  else print it out
  )
)

;*****************************************************************************
;                                                                             *
;  FUNCTION: writeout1                                                        *
;  PURPOSE:  To output a given item (atom, list, number, or string).          *
;                                                                             *
;*****************************************************************************

(define (writeout1 item)
;########################
 (cond
   ((listp item)   (print-list item 'space))  ; list? --> space at end
   (t (princ item) (princ " "))               ; prints out an atom,
 )                                            ;  a string, or a number
)

;*****************************************************************************
;                                                                             *
;  FUNCTION: blank                                                            *
;  PURPOSE:  To blank given number of lines                                   *
;                                                                             *
;*****************************************************************************

(define (blank number)
;######################
  (msg  (N number))
)

;*****************************************************************************
;                                                                             *
;  FUNCTION: space                                                            *
;  PURPOSE:  To print n blank spaces on a line.                               *
;                                                                             *
;*****************************************************************************

(define (space n)
;#################
 (cond ((eq n 1) (princ " "))
       (t (princ " ")
          (space (sub1 n))
       )
 )
)

;*****************************************************************************
;                                                                             *
;  FUNCTION: tell-user                                                        *
;  PURPOSE:  To output a message to the user with optional blanks lines       *
;            before and after the message, as well as optional spaces         *
```

```
;            before the line is begun.                                          *
;  INPUT:    a message and, optionally, # blanks before and after,              *
;            spaces before and punctuation.                                     *
;  OUTPUT:   Prints message in proper format to standard output.                *
;                                                                               *
;*******************************************************************************

(define (tell-user message
;##########################
         &optional numblanksbefore numblanksafter spacesbefore punctuation)
         ;=============================================================
 (cond ((numberp numblanksbefore) (blank numblanksbefore)))
 (cond ((numberp spacesbefore) (space spacesbefore)))
 (print-list message)
 (cond ((numberp numblanksafter) (blank numblanksafter)))
)

;*******************************************************************************
;                                                                               *
;  FUNCTION: open-input                                                         *
;  PURPOSE:  To open a port for input.                                          *
;  INPUT:    Name of the file to be generated.                                  *
;  OUTPUT:   Returns input port symbol.  Call by (setq $inport (open-input  *
;*******************************************************************************

(define (open-input filename $inport)
;####################################
 (setq $inport (infile filename))
)

(define (open-output filename $outport)
;######################################
 (setq $outport (outfile filename))
)

;-------------------------------------------------------------------------------
;                                                                               -
;  FUNCTION: all-but-last                                                       -
;  PURPOSE:  To return all but the last item in a given list.                   -
;  INPUT:    list                                                               -
;  OUTPUT:   list minus the last element                                        -
;                                                                               -
;-------------------------------------------------------------------------------

(define (all-but-last list)
;==========================
  (cond ((null (cdr list)) (cdr list))          ; tail empty --> ()
        (t (my-append (list (car list))         ; attach head of list to
                      (all-but-last (cdr list)) ; all but last of tail
           )
        )
  )
)

;-------------------------------------------------------------------------------
;                                                                               -
;  FUNCTION: my-append                                                          -
;  PURPOSE:  To append two lists together.                                      -
;  INPUT:    two lists                                                          -
;  OUTPUT:   first list appended to second list                                 -
;                                                                               -
;-------------------------------------------------------------------------------

(define (my-append list1 list2)
;==============================
 (cond ((null list1) list2)                     ; end of list1 - return list2
       (t (cons (car list1)                     ; else add 1st member of list1 to
                (my-append (cdr list1) list2)); result of recursing on tail
          )
       )                                        ;  of list1 and list2
 )
)

;-------------------------------------------------------------------------------
;  FUNCTION: print-list-commas-and                                              -
;  PURPOSE:  To print a list with commas between & word and before last.        -
;-------------------------------------------------------------------------------

(define (print-list-commas-and list-of-items)
 (cond
  ((null list-of-items) nil)
  ((eq (length list-of-items) 1)
   (msg (head list-of-items) ".")
  )
  ((eq (length list-of-items) 2)
   (msg (head list-of-items) " and " (head (tail list-of-items)) ".")
```

```
    )
    (t
     (commas-between (all-but-last list-of-items))
     (msg "and " (car (last list-of-items)) ".")
    )
   )
  )

(define (commas-between list-of-items)
 (mapcar 'print-item-with-comma list-of-items)
)

(define (print-item-with-comma item)
 (msg item ", ")
)
```

```
;*****************************************************************************
;                                                                            *
;  PROGRAM: STACK                                                            *
;  AUTHOR:  Mark Thomas Maybury                                              *
;  DATE:    November 17 1986                                                 *
;  PURPOSE: To keep track of stack operations.                               *
;                                                                            *
;  NB: stack is considered to be a global variable!                          *
;                                                                            *
;*****************************************************************************


;*****************************************************************************
;                                                                            *
;  FUNCTION: initialise-stack                                                *
;  PURPOSE:  To initialise a stack so that it can be used with my-push and   *
;            my-pop routines.  Note that unless you initialize a stack, you  *
;            may get unwanted errors.  For example, if you try to pop an     *
;            uninitilized stack before pushing anything on it, you will      *
;            get a lisp level error.                                         *
;  INPUT:    A symbol representing a stack.                                  *
;  OUTPUT:   Nothing, but side effect is setting stack to nil.               *
;                                                                            *
;*****************************************************************************

(defmacro initialise-stack (stack)
;#################################
  `(setq ,stack nil)  ; set to nil
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: my-push                                                         *
;  PURPOSE:  To push a given element onto a given stack.                     *
;            If the stack is empty, make it the list of give item.           *
;            Otherwise add the element to the current stack by using cons.   *
;  INPUT:    Item and stack                                                  *
;  OUTPUT:   Rebinds global variable.                                        *
;                                                                            *
;*****************************************************************************

(defmacro my-push (item stack)
;#############################
 `(setq ,stack (cons ,item ,stack)))

;*****************************************************************************
;                                                                            *
;  FUNCTION: my-pop                                                          *
;  PURPOSE:  To pop an item off a stack.                                     *
;  INPUT:    stack                                                           *
;  OUTPUT:   First element on list or nil if none.                           *
;                                                                            *
;*****************************************************************************

(defmacro my-pop (stack)
;#######################
 `(progl
   (car ,stack)
   (setq ,stack (cdr ,stack))
  )
)

(defmacro my-pop2 (stack)    ; does same thing somewhat less elegantly
;########################
 `(and
     (setq ts (car ,stack))
     (setq ,stack (cdr ,stack))
     ts
  )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: my-pop-result                                                   *
;  PURPOSE:  To pop an item off a stack and return the result.               *
;  INPUT:    stack                                                           *
;  OUTPUT:   Result of popping item from the list (ie cdr of list).          *
;                                                                            *
;*****************************************************************************

(defmacro my-pop-result (stack)
;##############################
  `(setq ,stack (cdr ,stack))
)
```

```
;*****************************************************************************
;                                                                             *
;  FUNCTION: empty?                                                           *
;  PURPOSE:  To determine if a stack is empty or not.                         *
;                                                                             *
;*****************************************************************************

(define (empty? stack)
;######################
 (if (null stack) then t else nil)
)

;*****************************************************************************
;                                                                             *
;  FUNCTION: peek-stack                                                       *
;  PURPOSE:  To peek at the top of the stack without altering its contents.   *
;                                                                             *
;*****************************************************************************

(define (peek-stack stack) (car stack))
;######################################
```

```
;////////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                               |
;  MODULE:   SAVE                                                               |
;  PURPOSE:  To aid in debugging by appending or writing a function to file.    |
;  OWNER:    copywrite Mark T. Maybury, July, 1987.                             |
;                                                                               |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////////


(define (save expr)
 (prog (fileout response)
  (msg N "Outfile:  " )                          ; get file for output
  (setq fileout (read))
  (msg "Append (Y/N): ")                         ; append to it?
  (setq response (read))
  (cond
   ((or (eq response 'Y) (eq response 'y))       ; user wants to append
    (msg N "Appending item")
    (setq fileout (outfile fileout 'a)))
   (t (setq fileout (outfile fileout))           ; else write over
      (msg "Item stored" N))
  )
  (terpri fileout)
  (terpri fileout)
  (pp-form expr fileout 2)                       ; left margin set at 2
  (close fileout)
 )
)
```

```
;////////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                               |
;  MODULE:  TRACK                                                               |
;  PURPOSE: To provide special tracing capabilities for the programmar.         |
;                                                                               |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////////

;*******************************************************************************
;                                                                               *
; FUNCTION:  TRACK                                                              *
; PURPOSE:                                                                      *
;            TRACK is a built-in debugging and trace mechanism which            *
;            allows the program developer and user to trace the functional      *
;            workings of the software.  This offers more flexibility and        *
;            greater ease of use (and more control) than the built-in           *
;            lisp TRACE function.  The level of tracking can be altered         *
;            with each run, dynamically allowing multiple levels of tracking    *
;            If level-desired is greater than threshold required for a          *
;            message to print out, then print it.  A blanking routine is        *
;            also provided.                                                     *
;                                                                               *
;   FUTURE --> let message be a body or executable code. (macro or fexpr?)      *
;                                                                               *
;*******************************************************************************

(define (track level-desired threshold message &optional blanks spaces-before)
 (cond
  ((>= level-desired threshold)                 ; above threshold?
   (cond (spaces-before (space spaces-before))) ; spaces?  spaces-before
   (print-list message)                         ;  print message
   (cond (blanks (blank blanks))                ;  blank blanks, if selected
         (t (blank 1))                          ;  default to 1
   )
  )
 )
)

(define (track-blank level-desired threshold number-of-blanks)
 (cond
  ((>= level-desired threshold)                 ; if above threshold
   (blank number-of-blanks)                     ; blank number of lines
  )
 )
)

(define (track-space level-desired threshold number-of-spaces)
 (cond
  ((>= level-desired threshold)                 ; if above threshold
   (space number-of-spaces)                     ; space number of spaces
  )
 )
)

; same as track but for functions to be evaluated

(define (trackf level-desired threshold function &optional blanks spaces-before)
 (cond
  ((>= level-desired threshold)                 ; above threshold?
   (cond (spaces-before (space spaces-before))) ; spaces?  spaces-before
   (eval function)                              ; evaluate function
   (cond (blanks (blank blanks))                ;  blank blanks, if selected
         (t (blank 1))                          ;  default to 1
   )
  )
 )
)

;-------------------------------------------------------------------------------
; FUNCTION: track-cpu                                                           |
; PURPOSE:  to collect run-time statistics for system efficiency evaluation.    |
;-------------------------------------------------------------------------------

(define (track-cpu time)
;-----------------------
 (cond
  ((eq time 'start) (ptime))                    ; begin tracking processor
  ((eq time 'finish)                            ; finish
   (let* ((cpu-time (ptime))
          (processor (first cpu-time))
          (garbage   (second cpu-time)))
    (track *tracing* 3 '(PROCESSING TIME) 1)
    (track *tracing* 3 '(CPU time used for processing: ,processor))
    (track *tracing* 3 '(CPU time used for garbage Collection: ,garbage))
   )
```

```
    )
  )
)
```

SECTION 4

KNOWLEDGE BASE INTERFACE

```
;////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                           |
;  MODULE:    KNOWLEDGE BASE INTERFACE                                      |
;  PURPOSE:   To provide access to the frame based knowledge representation |
;             where the neurophysiological and neuropsychological          |
;             knowledge is stored.                                          |
;  OWNER:     copywrite Mark T. Maybury, May, 1987.                         |
;                                                                           |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////

(include ~/dissert/KB/frames.lsp)          ; frame knowledge representaticn formalism
(include ~/dissert/KB/construct_kb.lsp)    ; kb generation routines
(include ~/dissert/KB/frame_access.l)      ; frame accessing primitives

; (include brain_kb.lsp)                   ; kb coLtaining brain knowledge
; (include disorder_kb.lsp)                ; kb containing disorder knowledge
```

```
;////////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                               |
;  MODULE:   FRAME KNOWLEDGE REPRESENTATION FORMALISM                           |
;  PURPOSE:  To provide a knowledge representation framework.                   |
;  OWNER:    copywrite Mark T. Maybury, June, 1987.                             |
;                                                                               |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////////

;*******************************************************************************
;    FRAME PROCEDURES FROM PATRICK HENRY WINSTON'S BOOK, LISP.  12 OCT 85      *
;*******************************************************************************

;*******************************************************************************
;                                                                              *
;   FGET retrieves information, given frame-slot-facet access path.            *
;                                                                              *
;*******************************************************************************

(define (fget frame slot facet)
   (cdr (assoc facet (cdr (assoc slot (cdr (get frame 'frame)))))))

;*******************************************************************************
;                                                                              *
;   FPUT places information.  given frame-slot-facet access path.              *
;                                                                              *
;*******************************************************************************

(define (fput frame slot facet value)
 (let ((value-list (follow-path (list slot facet)
                                (fget-frame frame))))
 (cond ((member value value-list) nil)
       (t (rplacd (last value-list) (list value))
          value))))

;*******************************************************************************
;                                                                              *
;   MY-FPUT places information like FPUT, given frame-slot-facet access path.  *
;     The difference, however, is that if you attempt to put the a piece of    *
;     information into a slot and that information is already there then       *
;     MY-FPUT returns the value of that piece of information whereas FPUT      *
;     returns NIL.                                                             *
;                                                                              *
;*******************************************************************************

(define (my-fput frame slot facet value)
 (let ((value-list (follow-path (list slot facet)
                                (fget-frame frame))))
 (cond (t (rplacd (last value-list) (list value))
          value)))
)

;*******************************************************************************
;                                                                              *
;   FGET-FRAME gets existing frame structure or creates one if non-existent    *
;                                                                              *
;*******************************************************************************

(define (fget-frame frame)
  (cond ((get frame 'frame))                          ; Frame already made?
        (t (setf (get frame 'frame) (list frame))))) ; If not, make one.

;*******************************************************************************
;                                                                              *
;   FREMOVE removes information, given frame-slot-facet access path.           *
;                                                                              *
;*******************************************************************************

(define (fremove frame slot facet value)
 (let ((value-list (follow-path (list slot facet)
                                (fget-frame frame))))
 (cond ((member value value-list)
        (delete value value-list)
        t)
       (t nil))))


;*******************************************************************************
;                                                                              *
;   EXTEND inspects first, using ASSOC, and if ASSOC fails, EXTEND extends     *
;          using RPLACD.                                                       *
;                                                                              *
;*******************************************************************************

(define (extend key a-list)
```

```
    (cond ((assoc key (cdr a-list));
          (t (cadr (rplacd (last a-list) (list (list key)))))))

;*****************************************************************************
;                                                                            *
;   FOLLOW-PATH uses EXTEND to push through frame structure.                 *
;                                                                            *
;*****************************************************************************

(define (follow-path path a-list)
  (cond ((null path) a-list)
        (t (follow-path (cdr path) (extend (car path) a-list)))))

;*****************************************************************************
;                                                                            *
;   FGET-V-D looks at VALUE facet of a given slot and then in the DEFAULT    *
;             facet if nothing is found in the VALUE facet.                  *
;                                                                            *
;*****************************************************************************

(define (fget-v-d frame slot)
  (cond ((fget frame 'value))
        ((fget frame slot 'default))))

;*****************************************************************************
;                                                                            *
;   FGET-V-D-P causes all procedures found in the IF-NEEDED facet to be      *
;              executed if neither VALUE nor DEFAULT facets help.            *
;                                                                            *
;*****************************************************************************

(define (fget-v-d-p frame slot)
  (cond ((fget frame slot 'value))        ; Try values first.
        ((fget frame slot 'default))      ; Then try defaults.
        (t (mapcar                        ; Combine results, if any.
           #'(lambda (demon) (funcall demon frame slot))
           (fget frame slot 'if-needed)))))

;*****************************************************************************
;                                                                            *
;   ASK could be a very popular occupant of the IF-NEEDED facet of a slot.   *
;                                                                            *
;*****************************************************************************

(define (ask frame slot)
 (print `(Please supply a value for the
                ,slot slot in the
                ,frame frame))      ;Start new line.
 (terpri)                           ;Get user's answer.
 (let ((response (read)))           ;Return list with answer if
      (t nil)))                     ;RESPONSE is other than NIL.

;*****************************************************************************
;                                                                            *
;   FGET-I uses FGET-CLASSES, a procedure that returns a list of all frames  *
;          that a given grame is linked to by an A-KIND-OF path, to give     *
;          values in frames related to given frame.                          *
;                                                                            *
;*****************************************************************************

(define (fget-i frame slot)
  (fget-il (fget-classes frame) slot))

(define (fget-il frames slot)
  (cond ((null frames) nil)                    ;Give up?
        ((fget (car frames) slot 'value))      ;Got something?
        (t (fget-il (cdr frames) slot))))      ;Climb tree

;*****************************************************************************
;                                                                            *
;   FPUT-P activate demons.                                                  *
;                                                                            *
;*****************************************************************************

(define (fput-p frame slot facet value)
 (cond ((fput frame slot facet value)
        (mapcar #'(lambda (e)                              ;Use procedures.
                    (mapcar #'(lambda (demon) (funcall demon frame slot))
                            (fget e slot 'if-added)))
        (fget-classes frame))
       VALUE)
 )
)
```

```
;*****************************************************************************
;                                                                            *
; CONSTRUCT-FRAME-KB takes a list of frames and uses MAKE-FRAME to make them *
;       frames when they are loaded in from a source file.                   *
;                                                                            *
;*****************************************************************************

(define (construct-frame-kb list-of-frames)
       (mapcar 'make-frame list-of-frames))

;*****************************************************************************
;                                                                            *
; MAKE-FRAME takes a particular FRAME and associates it with the name of     *
;     the particular frame, FRAME-NAME, the first element in the list FRAME. *
;                                                                            *
;*****************************************************************************

(define (make-frame frame)
 (let ((frame-name (car frame)))
  (setf (get frame-name 'frame) frame)))

;(define (add-on line1 line2)
; (strip-first-and-last-character line1)
; (concatenate line1 line2))
;
;(define (strip-first-and-last-character l)
; (implode (reverse (cdr (reverse (cdr explode l))))))
;
;
;(define (construct-frame-db)
;  (with-open-file (lobe-file 'lobe.lsp :dsk)
;       (prog ()
;        loop
;        (setq current-line (readline file 'end-of-file))
;        (if (not (equal current-line 'end-of-file))
;            (cond ((equal current-line '*) ((make-frame total-line)
;                                            (setq total-line nil)))
;                  (t (add-on currentline total-line))
;            )
;            (return ':end of file reached:))
;        (go loop)
;        ))
;)
;
```

```
;////////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                               |
;  MODULE:    FRAME ACCESSING                                                   |
;  PURPOSE:   To provide mnemonic primitives for frame access.                  |
;  OWNER:     copywrite Mark T. Maybury, June, 1987.                            |
;                                                                               |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////////


;-------------------------------------------------------------------------------
;                              Frame data retrieval                             |
;-------------------------------------------------------------------------------

;; family
(define (parents area) (fget area 'super-class 'value))
(define (children area) (fget area 'sub-class 'value))
(define (siblings area)
        (my-delete area (apply 'append (mapcar 'children (parents area)))))
;(define (children-type area) (fget area 'sub-class-type 'value))

;; attributes
(define (importance area) (integer->lex (first (fget area 'importance  'value))))
(define (dda area) (fget area 'dda 'value))  ; distinguishing descriptive attribute
(define (dda2 area) (mapcar 'seclist (fget area 'dda 'value)))
(define (dda3 area) (mapcar 'cdr (fget area 'dda 'value)))
(define (seclist dda) (list (first dda) (tail dda)))  ; (a b c) -> (a (b c))
(define (frame-type frame) (car (fget frame 'type 'value)))
(define (damage-of-area area) (integer->lex (car (reverse (fget area 'damage 'value)))))
(define (domain) (car (fget 'expert-domain 'sub-class 'value)))
(define (getscore frame slot facet)
        (car (reverse (fget frame slot facet))))

;; pragmatics
(define (relevance frame) (fget frame 'relevance 'value))
(define (context frame) (fget frame 'discourse-context 'value))

;-------------------------------------------------------------------------------
;                             Frame data manipulation                           |
;-------------------------------------------------------------------------------

;; attributes
(define (make-damage area value) (fput area 'damage 'value value))

;; focus
(define (place-in-vista frame) (my-fput frame 'relevance 'value 'in-vista))
(define (implicit-vista frame) (my-fput frame 'relevance 'value 'implicit-vista))
(define (mark-pragmatics area value) (fput frame 'pragmatics 'value value))
(define (place-in-context area purpose) (fput area 'discourse-context 'value purpose))

;-------------------------------------------------------------------------------
; FUNCTION: integer->lex                                                        |
; PURPOSE:  to convert an integer to a lexical entry.                           |
;-------------------------------------------------------------------------------

(define (integer->lex num)
;-------------------------
 (cond
  ((eq num 1) 'one)
  ((eq num 2) 'two)
  ((eq num 3) 'three)
  ((eq num 4) 'four)
  ((eq num 5) 'five)
  ((eq num 6) 'six)
  ((eq num 7) 'seven)
  ((eq num 8) 'eight)
  ((eq num 9) 'nine)
  ((eq num 10) 'ten)
 )
)
```

TEXT MODULE

```
;////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                           |
;  MODULE:    DISCOURSE SCHEMA                                              |
;  PURPOSE:   To make discourse decisions on what to say next constrained   |
;             by available knowledge (from global focus constraints), the   |
;             discourse goal, and local focus constraints.                  |
;             To instantiate rhetorical predicates and send them to tactical|
;             component.                                                    |
;  OWNER:     copywrite Mark T. Maybury, June, 1987.                        |
;             $Header: text.l,v 1.1 87/08/25 01:11:59 mtm Exp $             |
;                                                                           |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////

;---------------------------------------------------------------------------|
; FUNCTION: MAIN                                                            |
; PURPOSE:  to begin GENNY after initialization by determing the            |
;           discourse goal which is characterized by what we are going to   |
;           talk about (discourse focus) and how we are going to talk about |
;           it (discourse structure).                                       |
;                                                                           |
; LINGUISTIC                                                                |
; PRINCIPLES: This module exploits knowledge of common discourse strategies |
;             together with global and local focus constraints to generate  |
;             and then realise text for a provided discourse goal (e.g.     |
;             define, explain, compare) and discourse focus (e.g. frame).   |
;                                                                           |
;             First a theme-scheme is generated, built up by sub-schema     |
;             and their corresponding rhetorical predicates.  Next, a       |
;             vista of salient knowledge is selected from the knowledge     |
;             base, guided by the discourse topic.  Then, globally          |
;             constrained by this knowledge vista, a pool of relevant       |
;             propositions is generated.  GENNY then steps through the      |
;             theme-scheme, selecting propositions from the available pool   |
;             guided by a local focus model.  These are realised by a       |
;             tactical component which makes use of focus and context       |
;             to determine sentence structure and referring expressions,    |
;             and word choice (e.g. voice, anaphora, and articles).         |
;             The propositions are realised by a threefold process          |
;             including semantic interpretation, generation of relational   |
;             constituents, and building of a syntactic tree.  The final    |
;             surface form is determined by morphological and orthographic  |
;             procedures.  GENNY does not give up (or crash) if she fails   |
;             at any one of these stages.  Instead, she degrades gracefully |
;             by attempting to say anything that she can within the         |
;             boundries of the global and local constraints.)               |
;                                                                           |
;---------------------------------------------------------------------------|

(define (main)
;-------------
 (trackf *tracing* 2 '(welcome))                ; issue welcome and directions
 (load-dictionary)                              ; load domain dictionary
 (load-kb)                                      ; load domain knowledge base
 (trackf *tracing* 1 '(track-cpu 'start))       ; begin tracking cpu if requested
 (let* ((theme-scheme (discourse-scheme))       ; discourse schema
        (topic (discourse-topic theme-scheme)) ; discourse topic
        (discourse-structure (discourse theme-scheme topic))
        (realization (mapcar 'translate discourse-structure))
       )

  (track  *tracing* 2 '(DISCOURSE SCHEMA + FOCUS + GIVEN) 2)
  (trackf *tracing* 2 '(pp-form ',discourse-structure) 2)

  (track-blank *tracing* 1 3)
  (track  *tracing* 1 '(MESSAGE REALIZATION))
  (trackf *tracing* 1 '(pp-form ',realization) 2)

  (track-blank *tracing* 1 3)
  (track  *tracing* 1 '(SURFACE FORM))
  (msg (N 2))
  (mapcar 'surface-form realization)
  (trackf *tracing* 1 '(track-cpu 'finish))  ; finish tracking cpu if requested
 )
)

;---------------------------------------------------------------------------|
; FUNCTION: welcome                                                         |
; PURPOSE:  to print out welcome and directions                             |
;---------------------------------------------------------------------------|

(define (welcome)
;----------------
 (msg (N 3) "Welcome to the GENNY text generation system for expert systems.")
 (msg N     "GENNY was designed to answer questions of the form:")
```

```
 (msg (N 2) "-- What is an X?")
 (msg N     "-- Why did you diagnose Y?  or Why does Y have a problem?")
 (msg N     "-- What is the difference between X and Y?")
 (msg (N 2) "where X and Y are entities within the provided knowledge base.")
 (msg (N 2) "These three types of questions are indicated by the keywords:")
 (msg N     "DEFINE, EXPLAIN, and COMPARE, respectively.")
)

;-------------------------------------------------------------------------------
; FUNCTION: load-kb                                                            |
; PURPOSE:  to include a domain knowledge base.                                |
;-------------------------------------------------------------------------------

(define (load-kb)
;----------------
 (msg (N 2) "What is the domain of discourse?  ")
 (let ((file-with-kb (read)))
  (cond
   ((probef file-with-kb)                ; file exists?
    (load file-with-kb))
   (t (msg N "*** No file " file-with-kb " found." N))
  )
 )
)

;-------------------------------------------------------------------------------
;  FUNCTION: load-dictionary                                                   |
;  PURPOSE:  To load in a new dictionary, erasing the old one.                 |
;-------------------------------------------------------------------------------

(define (load-dictionary)
;------------------------
 (msg (N 2) "Please enter the domain dictionary file name?  ")
 (let ((file-with-dictionary (read)))
  (cond
   ((probef file-with-dictionary)               ; file exists?
    (erase-dictionary)                          ; defined in makedictionary
    (load file-with-dictionary))
   (t (msg N "*** No file " file-with-dictionary " found." N))
  )
 )
)

;-------------------------------------------------------------------------------
; FUNCTION: discourse-scheme                                                   |
; PURPOSE:  to determine thematic-scheme or text sketch for answer.            |
;-------------------------------------------------------------------------------

(define (discourse-scheme)
;-------------------------
 (msg (N 2) "Do you wish DEFINE, EXPLAIN, or COMPARE?  ")
 (get-a-discourse-goal)
)

;-------------------------------------------------------------------------------
; FUNCTION: get-a-discourse-goal                                               |
; PURPOSE:  to query the user for a frame name in the current KB.              |
;-------------------------------------------------------------------------------

(define (get-a-discourse-goal)
;-----------------------------
 (let ((response (read)))
  (cond
   ((fget-frame response) response)
   (t (msg (N 2) "GENNY cannot " response ".")
      (msg N "Please type another reponse (DEFINE/EXPLAIN/COMPARE):  ")
      (get-a-discourse-goal)
   )
  )
 )
)
;-------------------------------------------------------------------------------
; FUNCTION: discourse-topic                                                    |
; PURPOSE:  to determine focus (foci, if comparison) of attention of text.     |
;-------------------------------------------------------------------------------

(define (discourse-topic theme-scheme)
;-------------------------------------
 (cond
  ((eq theme-scheme 'compare)
   (msg N "What do you wish to compare?  ")
   (let ((entity (get-a-frame-name)))
    (msg N "What would you like to compare it to?  ")
    (list entity (get-a-frame-name))
   )
  )
```

```
    (t
     (msg (N 2) "What do you wish to know about?  ")
     (list (get-a-frame-name))
    )
   )
  )

;-------------------------------------------------------------------------------
; FUNCTION: get-a-frame-name                                                    |
; PURPOSE:  to query the user for a frame name in the current KB.               |
;-------------------------------------------------------------------------------

(define (get-a-frame-name)
;-------------------------
 (let ((response (read)))
  (cond
   ((fget-frame response) response)
   (t (msg (N 2) "GENNY has no knowledge about " response ".")
      (msg N "Please type another reponse:  ")
      (get-a-frame-name)
   )
  )
 )
)

;-------------------------------------------------------------------------------
; FUNCTION: discourse                                                           |
; PURPOSE:  To generate a specific text structure given a TS (a text            |
;           structure or a theme-scheme) along with an item.                    |
;-------------------------------------------------------------------------------

(define (discourse scheme item)
;------------------------------
 (let* ((global-focus item)
        (ts (thematic-scheme scheme global-focus))
        (kvista (select-knowledge-vista global-focus)) ; vista into relevant knowledge
       )

  (clear-relevant-propositions *predicate-types*)

  (track-blank *tracing* 1 1)
  (track *tracing* 1 '(SELECT KNOWLEDGE VISTA ==> ,kvista) 2)

  (track  *tracing* 1 '(GENERATE RELEVANT PROPOSITION POOL) 2)

  (generate-relevant-propositions            ; generate relevant propositions
     kvista
     *predicate-types*
     scheme)

  (track  *tracing* 1 '(GENERATE DISCOURSE PLAN:))
  (trackf *tracing* 1 '(pp-form ',ts) 2)

  (track  *tracing* 1 '(GLOBAL FOCUS (DISCOURSE TOPIC) ==> ,global-focus) 2)

  (generate-discourse               ; generate discourse propositions
    ts                              ;  discourse plan
    nil                             ;  no past foci
    global-focus                    ;  current focus = global focus(i)
    nil                             ;  no knowledge of potential future foci
    nil)                            ;  no current context
 )
)

;(defmacro makelistifnot (l) '(cond ((listp ,l) ,l) (t (list ,l))))

;-------------------------------------------------------------------------------
; FUNCTION: clear-relevant-propositions                                         |
; PURPOSE:  to clear the instantiated propositions from each predicate's        |
;           property list.                                                      |
;-------------------------------------------------------------------------------

(define (clear-relevant-propositions predicates)
;-----------------------------------------------
 (cond
  ((null predicates))
  (t (putprop (head predicates) nil 'propositions)        ; clear propositions
     (clear-relevant-propositions (tail predicates))      ; tail recurse
  )
 )
)

;-------------------------------------------------------------------------------
; FUNCTION: propositions                                                        |
; PURPOSE:  to return the instantiated propositions in the pool of              |
;           knowledge for a given rhetorical predicate.                         |
```

```
;-------------------------------------------------------------------------

(define (propositions predicate) (get predicate 'propositions))
;-------------------------------------------------------------------

;-------------------------------------------------------------------------
; FUNCTION: generate-discourse                                           |
; PURPOSE:  to take a plan of discourse predicates along with focus      |
;           information and recursively generate a list of rhetorical    |
;           propositions.  Local focus constraints are used to constrain |
;           choice at any particular juncture in the thematic-scheme and |
;           global focus constraints limit the propositions which are    |
;           successfully instantiated or matched against the vista in the|
;           knowledge base.                                              |
;-------------------------------------------------------------------------

(define (generate-discourse thematic-scheme pf cf ff context)
;-------------------------------------------------------------

 (cond
  ((null thematic-scheme) nil)                ; nothing more to talk about
  (t
   (let* ((next-illocutionary-action+focus    ; choose next thing to talk about
            (select-proposition
             (propositions (first thematic-scheme))
             pf cf ff))
          (next-illocutionary-action (head next-illocutionary-action+focus))
          (next-foci (tail next-illocutionary-action+focus))
          (DF (first next-foci))              ; default foci of next predicate
          (AF (second next-foci))             ; alternate foci of next predicate
         )


;    (track  *tracing* 2 '(NEXT DISCOURSE PROPOSITION:))
;    (trackf *tracing* 2 '(pp-form ',next-illocutionary-action) 2)
;    (track  *tracing* 2 '(CURRENT CONTEXT (GIVEN):))
;    (trackf *tracing* 2 '(pp-form ',context) 2)

    (append
     (cond ((null next-illocutionary-action) nil)   ; if next proposition then
           (t (list (list next-illocutionary-action ; save proposition
                          (list pf DF AF)           ; save focus information
                          context))))               ; return context
     (generate-discourse
      (tail thematic-scheme)                        ; choose rest to say
       (cond
             ((tail cf)                            ;(> (length cf) 1);multiple foci?
              (append (list DF)                     ;  last uttered
                      (list (tail cf))              ;  not yet utt
                      pf))                          ;  other past
             ((member (car DF) ff)                   ; DF in ff? (multiple focus?)
               (append (list DF)                     ;  past = push ff on pf
                       (list (delete (car DF) ff))
                       pf))
              (t (append (list DF) pf)))            ;  past = push DF on pf
       (cond ((tail cf) cf)
             (t DF))
       AF                                           ; future foci
       (delete-duplicates (append DF AF context))   ; save current context
      )
     )
    )
   )
  )
 )
)

;-------------------------------------------------------------------------
; FUNCTION: generate-relevant-propositions                               |
; PURPOSE:  to generate a set of relevant propositions from the provided |
;           knowledge vista.                                             |
;-------------------------------------------------------------------------

(define (generate-relevant-propositions kvista predicate-types speech-act)
;-------------------------------------------------------------------------

 (cond
  ((null predicate-types))                                        ; stop
  (t
   (putprop (head predicate-types)                                ; save relevant
            (match-predicate (head predicate-types) kvista speech-act);  propositions
            'propositions)
   (generate-relevant-propositions
      kvista (tail predicate-types) speech-act) ; tail recurse
  )
 )
)

;-------------------------------------------------------------------------
```

```
; FUNCTION: match-predicate                                                    |
; PURPOSE:  to match a predicate against the vista in the knowledge base.     |
;------------------------------------------------------------------------------

(define (match-predicate type kvista speech-act)
;-----------------------------------------------
 (cond
  ((null kvista) nil)                                    ; no more knowledge
  (t
   (let ((next-pred (instantiate-predicate               ; try to make next pred
                      type
                      (head kvista)
                      speech-act)))
    (append                                              ; else construct
      (cond ((null next-pred) nil)                       ;  first item in kvista
            (t (list next-pred)))                        ;   if any
      (match-predicate type (tail kvista) speech-act)) ;  rest items in kvista
   )
  )
 )
)

;------------------------------------------------------------------------------
; FUNCTION: thematic-scheme                                                    |
; PURPOSE:  to return a sketch structure of the text provided with the theme|
; METHOD:   uses discourse primitives (such as attributive, constituent,)     |
;           which provide a framework for the given global discourse goal    |
;           (eg define, explain).  Note the global discourse goal is a type  |
;           of speech act (ref Searle).  A speech acts planning module       |
;           could be added here a la Paul Cohen.                             |
;------------------------------------------------------------------------------

(define (thematic-scheme theme topic)
;------------------------------------
 (track-blank *tracing* 1 3)
 (track  *tracing* 1 '(TEXT SKETCH:) 2)
 (cond
  ((eq theme 'define)
   (append
    (sub-schema 'introduction topic)
    (sub-schema 'description topic)
    (sub-schema 'example topic)
   )
  )
  ((eq theme 'explain)
   (append
    (sub-schema 'reason topic)
    (sub-schema 'evidence topic)
   )
  )
  ((eq theme 'compare)
   (append
    (sub-schema 'introduction topic)
    (sub-schema 'introduction topic)
    (sub-schema 'comparison topic)
    (sub-schema 'conclusion topic)
   )
  )
  (t (msg N "Sorry, but GENNY has no knowledge of the " theme " theme-scheme" N))
 )
)

;------------------------------------------------------------------------------
; FUNCTION: sub-schema                                                         |
; PURPOSE:  to return the sketch of a sub-discourse given a                    |
;           perlocutionary-act.                                                |
;------------------------------------------------------------------------------

(define (sub-schema perlocutionary-act topic)
;--------------------------------------------
 (trackf *tracing* 1 '(pp-form ',perlocutionary-act) 1)
 (cond
  ((eq perlocutionary-act 'introduction)
   '(definition attributive))
  ((eq perlocutionary-act 'description)
   (append '(constituent)
     (option
      (predicate* 'attributive topic)
      '(definition)
     )
   )
  )
  ((eq perlocutionary-act 'example) '(illustration))
  ((eq perlocutionary-act 'reason) '(cause-effect))
  ((eq perlocutionary-act 'evidence)
   (option
```

```
     (predicate* 'attributive topic)
     '(definition)
    )
   )
   ((eq perlocutionary-act 'comparison)  '(compare-contrast))
   ((eq perlocutionary-act 'conclusion) '(inference))
  )
 )

;-------------------------------------------------------------------------
; FUNCTION: option                                                       |
; PURPOSE:  to allow a choice between a first or second item.  If none ->nil|
;-------------------------------------------------------------------------

(define (option first second)
;----------------------------
 (cond
  ((null first) second)                    ; if first empty, return second
  (t first)                                ; else return the first
 )
)

;-------------------------------------------------------------------------
; FUNCTION: predicate*                                                   |
; PURPOSE:  to allow for multiple repetition of a predicate              |
;-------------------------------------------------------------------------

(define (predicate* predicate topic)
;-----------------------------------
 (let ((childs (length (children (first topic)))))
  (cond
   ((zerop childs) nil)                          ; no children? -> stop
   (t (repeat predicate childs))                 ; else repeat the predicate
  )                                              ;  for each child
 )
)

;-------------------------------------------------------------------------
; FUNCTION: repeat                                                       |
; PURPOSE:  to repeat a given symbol n-times by ingenious use of array   |
;           function for duplication (see Wilensky, LISPcraft, 1984 for  |
;           descriptions of array functions).                            |
;-------------------------------------------------------------------------

(define (repeat symbol n-times)
;------------------------------
 (let ((temp-array (newsym 'array)))     ; newsym for temporay local variable
  (eval '(array ,temp-array t n-times))  ; define local array
  (fillarray temp-array (list symbol))   ; fill array with symbol
  (listarray temp-array n-times)         ; list array out to nth element
 )
)
```

SECTION 6

RHETORICAL PREDICATE SEMANTICS

```
;////////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                               |
;  MODULE:    PREDICATE SEMANTICS                                               |
;  PURPOSE:   To associate the individual rhetorical predicates with the        |
;             frame based knowledge representation.                             |
;  NOTE:      While these semantics are domain-independent, easily ported       |
;             to any domain represented in a frame network, they are            |
;             knowledge representation specific.  Porting the generator         |
;             to another KR scheme (logic, for example) would require a new     |
;             semantic link into that representation.  Hence, this module       |
;             would be replaced.                                                |
;  OWNER:     copywrite Mark T. Maybury, June, 1987.                            |
;                                                                               |
;  LINGUISTIC                                                                   |
;  MOTIVATION:  It seems that discourse can be elegantly described in           |
;               terms of rhetorical acts or predicates which serve as           |
;               text-type independent building blocks of larger discourse.      |
;               This stems from work by early grammarians (c.f. Williams,       |
;               1893), more recently investigated by Grimes (1975), with        |
;               related speech-act work by Searle (1969, 1975).  McKeown's      |
;               (1985) work, a major contribution to text structure             |
;               definition, motivates these predicates, although the            |
;               discourse goals in GENNY include explanations in addition       |
;               to definitions and comparisons.  More importantly, the          |
;               predicates presented below -- definition, attributive,          |
;               constituent, evidence, illustration, cause-effect,              |
;               compare-contrast, and inference -- are interfaced to a          |
;               frame knowledge formalism, as defined by the following          |
;               predicate semantics.                                            |
;                                                                               |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////////


;;;   RHETORICAL PREDICATE + SEMANTICS + KNOWLEDGE BASE ====> PROPOSITION

;-------------------------------------------------------------------------------
; FUNCTION: instantiate-predicate                                               |
; PURPOSE:  to return a proposition which is an instantiation of a              |
;           rhetorical predicate with knowledge from the frame KB.              |
;-------------------------------------------------------------------------------

(define (instantiate-predicate type frame speech-act)
;----------------------------------------------------
 (cond
  ((atom frame)                                    ; if single foci
   (cond
    ((and (frame-type frame)                    ; does the frame
          (head (parents frame)))               ; exist/not root node?

       ; PREDICATE DOCUMENTATION GUIDE:
       ;  PREDICATE-NAME
       ;  semantics
       ;  instantiated example
       ;  realization1
       ;  realization2

     (cons                                                ; OUTPUT ==
      type                                                ;   predicate-type
      (cond                                               ; + predicate

       ; DEFINITION PREDICATE
       ; [definition type/entity type dda]
       ; [definition ((region brain)) ((region)) ((location (skull))))]
       ; The brain region is a region located in the skull.
       ; There is a region located in the skull called the brain region.

       ((eq type 'definition)
        (list
         (cond
          ((or (act? frame) (result? frame))
             `((,(frame-type frame) ,frame)))
          (t `((,frame))))
         `((,(frame-type frame)))
          (dda2 frame)
        )
       )

       ; ATTRIBUTIVE PREDICATE
       ; [attributive type/entity (attr value)*]
       ; [attributive ((region brain)) ((value importance indef 1))]
       ; The brain region has an importance value of 1.
       ; An importance of 1 is attributed to the brain region.

       ((eq type 'attributive)
```

```
                        (list
                        '((,(frame-type frame) ,frame))
                         (attributes frame speech-act)
                        )
                       )

                       ; CONSTITUENT PREDICATE
                       ; [constituent entity sub-class-type sub-class-no subclasses]
                       ; [constituent ((brain)) ((hemisphere two none)) nil ((l-hem region) (r-hem region)
)]
                       ; The brain has two hemispheres:  a l-hem region and a r-hem region.
                       ; There are two hem in the brain: a l-hem region and a r-hem region.

                       ((eq type 'constituent)
                        (list
                         '((,frame))
                         (cond
                          ((children frame)                          ; if there are children
                           (list (list
                            (frame-type (first (children frame)))
                            (integer->lex (length (children frame))) 'none)))
                          (t nil))
                         nil                                         ; no instrument, function,
                         (mapcar 'type+frame (children frame))       ;  or location
                        )
                       )                                                    ;'condition

                       ; EVIDENCE PREDICATE
                       ; [evidence (type entity) '((damage)) '((location (super-type super-class)))]
                       ; [evidence ((test language)) ((damage)) ((location (lobe lfrontal)))] ]
                       ; The language (test) indicates the damage in
                       ;   the lfrontal lobe. (Language indicates damage in lfrontal?)
                       ; The lfrontal lobe damage is indicated by the language test.
                       ;

                       ((eq type 'evidence)
                        (list
                         (cond
                          ((or (act? frame) (result? frame))
                              '((,(frame-type frame) ,frame)))
                          (t '((,frame))))
                         '((damage))
                         (cond
                          ((or (act? (first (parents frame)))
                               (result? (first (parents frame))))
                           (list (list
                            'instrument
                            (list (frame-type (first (parents frame)))
                                  (first (parents frame)))))
                           )
                           (t
                            (list (list
                             'location
                             (list (frame-type (first (parents frame)))
                                   (first (parents frame)))))
                           )
                          )
                         )
                       )

                       ; ILLUSTRATION PREDICATE
                       ; [illustration type/entity type dda]
                       ; [illustration ((region left-hemiphere)) ((function (feature-recognizer))) brain ]
                       ; The l-hem region, for example, functions as a feature-reconizer for the brain.
                       ; The l-hem region, for ex, has the feature-recognizer function in the brain

                       ((eq type 'illustration)
                        (list
                         '((,(frame-type frame) ,frame))
                         '((,(frame-type frame)))
                         (dda2 frame)
                        )
                       )

                       ; CAUSE-EFFECT PREDICATE
                       ; [cause-effect entity '((damage)) sub-class '((damage))]
                       ; [cause-effect ((brain reg)) '((damage)) nil ((l-hem reg) (r-hem reg)) '((damage))
]
                       ; The brain region has (no) damage because the l-hem region and the r-hem region ha
ve (no) damage.
                       ; The amnesic disorder is manifest because the apathetic observation indicates dama
ge.

                       ((eq type 'cause-effect)
                        (list
                        '((,(frame-type frame) ,frame))
```

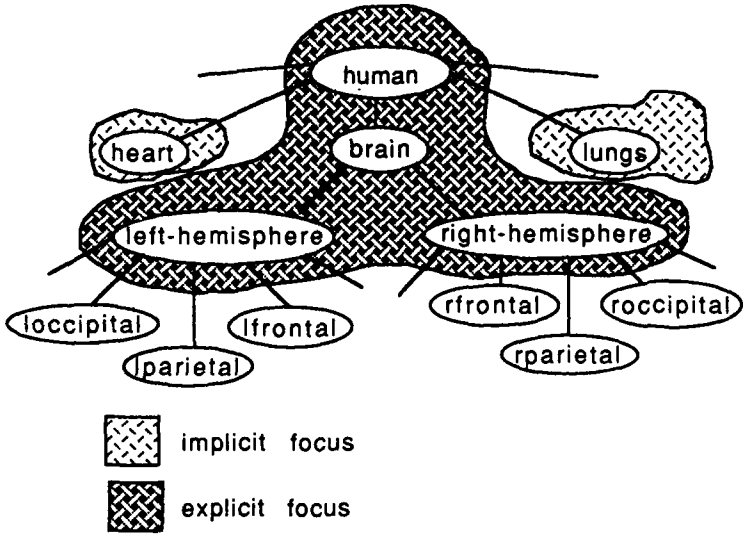
```
                    (cond
                     ((result? frame) '((manifest)))   ; is frame a result (i.e. symptom/disorder)
                     ((act? frame) '((made)))          ; is frame an act (i.e. observation)
                     (t '((damaged nil none))))        ; else it is an object
                    nil                                ; no instrument, function, location
                    (mapcar 'type+frame (children frame))
                   '((damage))
                   )
                  )
                 )
                )
               )
              )
             )
            ((listp frame)                                    ; make sure its a list
             (cond
              ((and                                           ; if multiple foci frame
                (and (frame-type (first frame)) (head (parents (first frame))))   ; f1
                (and (frame-type (second frame)) (head (parents (second frame)))) ; f2
               )
               (cons                                          ; OUTPUT ==
                type                                          ;   predicate-type
                (cond                                         ; + predicate

                 ; COMPARE-CONTRAST PREDICATE
                 ; [compare-contrast (entity1 entity2) (comparison val)*]
                 ; [compare-contrast ((1-hem) (r-hem)) ((dda similar) (type different))]
                 ; The 1-hem and the r-hem have a similar type and a different dda.
                 ; There is a similar type and a different dda for the 1-hem and the r-hem.

                 ((eq type 'compare-contrast)
                  (list
                   (list
                    (list (first frame))
                    (list (second frame)))
                   (comp-cont (first frame) (second frame)))
                  )

                 ;  INFERENCE PREDICATE
                 ;  [inference frame1 frame2 conclusion]
                 ;  [inference ((brain) (language)) ((entity different none))]
                 ;  The brain and language, therefore, are different entities.
                 ;  Hence, the brain is different from language.

                 ((eq type 'inference)
                  (list
                   (list
                    (list (frame-type (first frame)) (first frame))
                    (list (frame-type (second frame)) (second frame)))
                   '((entity
                     ,(inference (first frame) (second frame)) none)))
                  )
                 )
                )
               )
              (t nil)                                         ; if frame doesn't exist -> nil
             )
            ) ; listp
           )
          )

          ;-------------------------------------------------------------------------
          ; FUNCTION:   mark-as-used                                               |
          ; PURPOSE:    to mark one or a number of frames as used for a particular |
          ;             rhetorical purpose.  This device acts as a discourse context|
          ;             which records past utterances.  This can aid in resolving  |
          ;             focus selection when such a choice is ambiguous.           |
          ; THEORETICAL                                                           |
          ; MOTIVATION People don't normally repeat themselves in discourse unless |
          ;             they wish to achieve a peculiar effect (eg emphasis,       |
          ;             conversational implicature, etc).  Thus, record past usages|
          ;             of a particular knowledge chunk so that is not repeated    |
          ;             later in the discourse.                                    |
          ;                                                                        |
          ; MECHANISM: The frame slot called "discourse-context" is marked with the |
          ;             symbol representing the rhetorical predicate type          |
          ;             (eg illustrative or constituent).  When the pool of knowledge|
          ;             is being constructed, this field is tested and no proposition|
          ;             is generated for a particular rhetorical predicate if that |
          ;             utterance has already occured in the discourse.            |
          ;-------------------------------------------------------------------------

          (define (mark-as-used frames predicate)
          ;--------------------------------------
           (cond
            ((null frames))
```

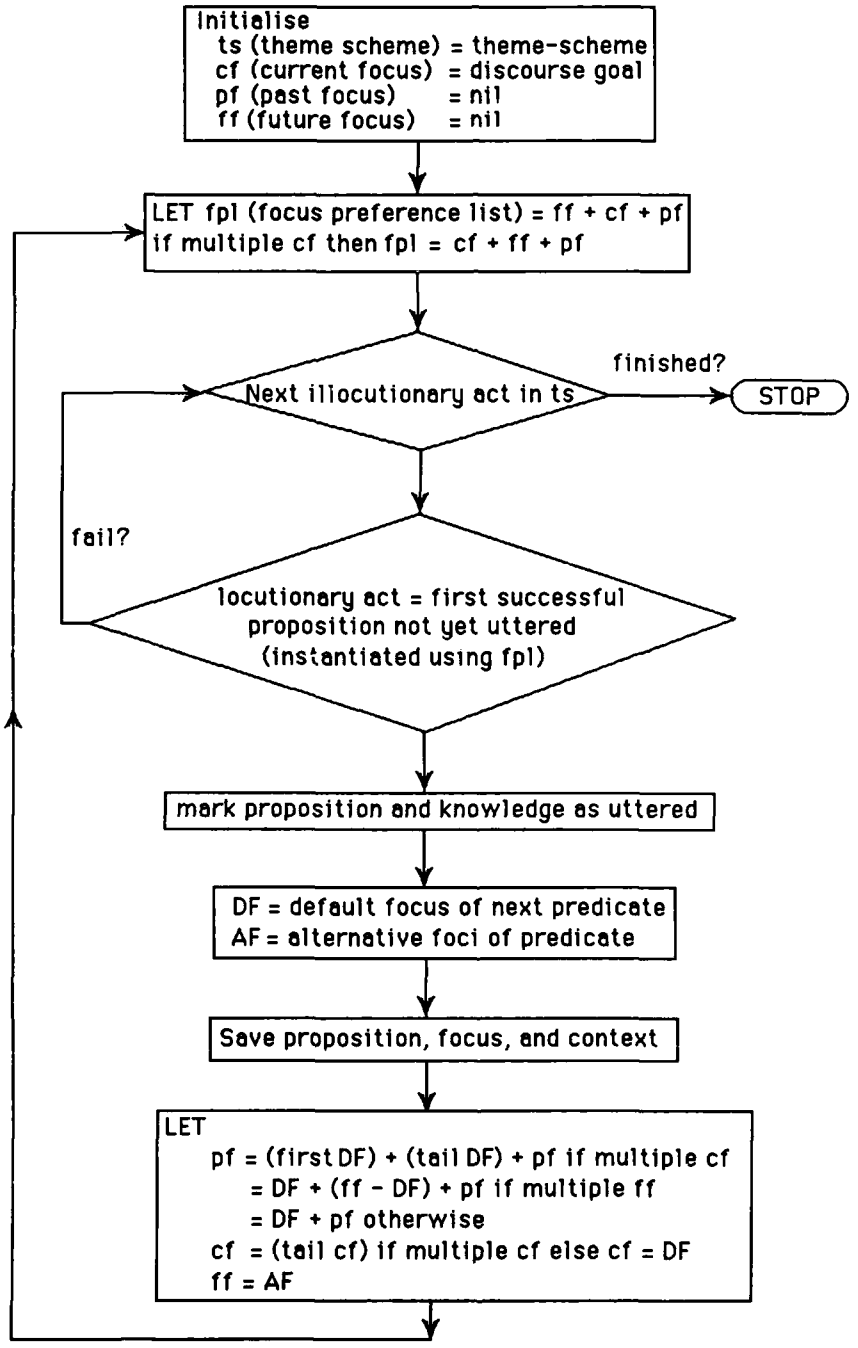
```
          ((atom frames) (place-in-context frames predicate))   ; if one, in context
          (t (place-in-context (head frames) predicate)         ; else first in context
             (mark-as-used (tail frames) predicate))            ;  and rest
         )
        )

        ;-------------------------------------------------------------------------------
        ; FUNCTION: type+frame                                                         |
        ; PURPOSE:  to return a list of the frame name together with its type.         |
        ;-------------------------------------------------------------------------------

        (define (type+frame frame)
        ;--------------------------
         (list (frame-type frame) frame))

        ;-------------------------------------------------------------------------------
        ; FUNCTION: result?                                                            |
        ; PURPOSE:  to determine if a given frame is a resultant of something.         |
        ;-------------------------------------------------------------------------------

        (define (result? frame)
        ;-----------------------
         (cond ((member (frame-type frame) '(symptom disorder fault attribute))))
        )

        ;-------------------------------------------------------------------------------
        ; FUNCTION: act?                                                               |
        ; PURPOSE:  to determine is the provided frame is an act.                      |
        ;-------------------------------------------------------------------------------

        (define (act? frame)
        ;--------------------
         (cond ((member (frame-type frame) '(observation))))
        )

        ;-------------------------------------------------------------------------------
        ; FUNCTION: attributes                                                         |
        ; PURPOSE:  to return the attributes of the given frame by examining KB.       |
        ;-------------------------------------------------------------------------------

        (define (attributes frame speech-act)
        ;-------------------------------------
         (let ((import (importance frame))
               (damage (damage-of-area frame))
               (type   (frame-type frame)))

          (cond
           ((eq speech-act 'define)
            (cond ((null import) nil)
                  (t (list (list 'value 'importance 'indef import 'relative)))))
           ((eq type 'test)
            (cond ((null damage) nil)
                  (t (list (list 'result nil 'indef damage)))))
           ((or (eq type 'symptom) (eq type 'observation))
            (cond ((null damage) nil)
                  (t (list (list 'value 'likelihood 'indef damage)))))
           ((eq speech-act 'explain)
            (cond ((null damage) nil)
                  (t (list (list 'value 'damage 'indef damage)))))
           ((eq speech-act 'compare)
            (cond ((and (null import) (null damage)) nil)
                  ((null import) (list (list 'value 'damage 'indef damage)))
                  ((null damage) (list (list 'value 'importance 'indef import 'relative)))
                  (t (list (list 'value 'importance 'indef import 'relative)
                           (list 'value 'damage 'indef damage)))
            )
           )
          )
         )
        )
        ;  may actually want this form which returns,
        ;  for ex, ((importance 1) (damage nil)) versus ((importance 1))
        ;  so can say "The brain has importance of 1 and damage of unknown."
        ;             "The brain has a 0.8 importance and an unknown damage."
        ;  (list
        ;   (list 'importance import)
        ;   (list 'damage     damage)
        ;   (list 'sub-class  childs)
        ;  )

        ;-------------------------------------------------------------------------------
        ; FUNCTION: inference                                                          |
        ; PURPOSE:  to make inference on two provided frames.                          |
        ;-------------------------------------------------------------------------------

        (define (inference f1 f2)
```

```
        ;-------------------------
         (cond
          ((eq f1 f2) 'equal)
          ((eq (frame-type f1) (frame-type f2)) 'similar)
          ((eq (parents f1) (parents f2)) 'similar)
          ((eq (dda f1) (dda f2)) 'similar)
          (t 'different)
         )
        )

        ;-----------------------------------------------------------------------------
        ; FUNCTION: comp-cont                                                        |
        ; PURPOSE:  to compare or contrast to given frames.                          |
        ; METHOD:   Check equality of various slot values.                           |
        ;           Check hierarchical distance.                                     |
        ;           Check similarity of parents, children, siblings.                 |
        ;-----------------------------------------------------------------------------

        (define (comp-cont f1 f2)
        ;-------------------------
         (list
        ;  (list 'name (cond ((eq f1 f2) 'equal) (t 'different)))
           (list 'class (cond ((eq (parents f1) (parents f2)) 'similar) (t 'different)))
        ;  (list 'sub-class (cond ((eq (children f1) (children f2)) 'similar) (t 'different)))
           (list 'type (cond ((eq (frame-type f1) (frame-type f2)) 'similar) (t 'different)))
        ;  (list 'dda (cond ((eq (dda f1) (dda f2)) 'similar) (t 'different)))
           (list 'importance (cond ((eq (importance f1) (importance f2)) 'similar) (t 'different))
)
         )
        )
```

SECTION 7

FOCUS AND ANAPHOR ALGORITHMS


human
heart
brain
lungs
left-hemisphere
right-hemisphere
loccipital
lfrontal
lparietal
rfrontal
roccipital
rparietal
implicit focus
explicit focus


Initialise
ts (theme scheme) = theme-scheme
cf (current focus) = discourse goal
pf (past focus) = nil
ff (future focus) = nil
LET fpl (focus preference list) = ff + cf + pf
if multiple cf then fpl = cf + ff + pf
Next illocutionary act in ts
finished?
STOP
fail?
locutionary act = first successful proposition not yet uttered (instantiated using fpl)
mark proposition and knowledge as uttered
DF = default focus of next predicate
AF = alternative foci of predicate
Save proposition, focus, and context
LET
pf = (first DF) + (tail DF) + pf if multiple cf
= DF + (ff - DF) + pf if multiple ff
= DF + pf otherwise
cf = (tail cf) if multiple cf else cf = DF
ff = AF

```
;////////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                               |
;  MODULE:    FOCUS                                                             |
;  PURPOSE:   To determine and utilize both global and local focus              |
;             constraints to enhance relevancy of knowledge and                 |
;             connectivity of discourse.                                        |
;  OWNER:     copywrite Mark T. Maybury, June, 1987.                            |
;             $Header: focus.l,v 1.1 87/08/25 01:09:59 mtm Exp $                |
;                                                                               |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////////


;===============================================================================
;                                                                               |
;                              GLOBAL FOCUS THEORY                              |
;                              -------------------                              |
;                                                                               |
;  Motivated by Barbara Gross's theory of global focus, declare a knowledge     |
;  vista which encompases the given frame, its parent, and children.  A         |
;  more sophisticated mechanism (beyond the scope of this dissertation)         |
;  could incorporate user modelling to select relevant knowledge with           |
;  regard to the level of sophistication of the audience.  Furthemore, a        |
;  discourse model could supress or encourage certain peices of information     |
;  with regard to previously generated text.                                    |
;                                                                               |
;  The use of the diction "knowledge vista" illustrates the connection with     |
;  the FRL approach to knowledge representation.  In this paradigm,             |
;  knowledge is most perpicuous when viewed from some relevant vista.           |
;  Here, knowledge in global focus is the relevant vista.                       |
;                                                                               |
;===============================================================================

;-------------------------------------------------------------------------------
; FUNCTION: select-knowledge-vista                                              |
; PURPOSE:  to select a vista within the knowledge base which reflects the      |
;           knowledge relevant to the global-focus(i).                          |
; METHOD:   utilize the frame hierarchy to place the superordinate and          |
;           subordinate classes of the frames into the knowledge vista.         |
;           Also, siblings of frames are placed in implicit vista or focus.     |
; LINGUISTIC                                                                    |
; THEORY:   Humans place not only individual entities but also multiple         |
;           entities in focus simultaneously.  This is the case, for            |
;           example, when conversants discuss two items in parallel, as in      |
;           comparison.                                                         |
;-------------------------------------------------------------------------------

(define (select-knowledge-vista frames)
;--------------------------------------
 (let
  ((frames-vista                                ; frames vista consists of
    (append
     frames                                     ;   frames in focus
     (apply 'append (mapcar 'children frames));   their children
     (apply 'append (mapcar 'parents frames)) ;   their parents
    )
   )
  )
  (mapcar 'place-in-vista frames-vista)         ; mark knowledge vista
  (mapcar 'implicit-vista
   (apply 'append (mapcar 'siblings frames)))   ; mark implicit knowledge vista
                                                ;  (could mark two levels away)
  (cond
   ((listp frames) (cons frames frames-vista)); multiple global foci? add to kvista
   (t frames-vista)                             ; return kvista
  )
 )
)


;-------------------------------------------------------------------------------
; FUNCTION: in-vista?                                                           |
; PURPOSE:  to determine if a given frame is in global focus (in K vista).      |
;-------------------------------------------------------------------------------

(define (in-vista? frame)
;------------------------
 (cond ((eq (relevance frame) 'in-vista))))

;-------------------------------------------------------------------------------
; FUNCTION: implicit-vista?                                                     |
; PURPOSE:  to determine if a given frame is implicitly in focus.               |
;-------------------------------------------------------------------------------

(define (implicit-vista? frame)
```

```
;------------------------------
 (cond ((eq (relevance frame) 'implicit-vista))))


;=============================================================================
;                                                                            |
;                              LOCAL FOCUS THEORY                            |
;                              ------------------                            |
;                                                                            |
;  PF  == past foci list                                                     |
;  CF  == current focus                                                      |
;  FF  == potential future foci list                                         |
;                                                                            |
;  Attentional shift algorithm (motivated by Candace Sidner's work).        |
;                                                                            |
;  if choice between PF or CF, prefer CF  (to continue present topic)        |
;  if choice between CF or FF, prefer FF  (to introduce new topics)          |
;                                                                            |
;  INSIGHT                                                                   |
;  if possible, stick to topic unless future role is illustration, then FF   |
;  otherwise, allow digression from topic for one level,                     |
;             then return to topic.                                          |
;                                                                            |
;=============================================================================

;-----------------------------------------------------------------------------
; FUNCTION: select-proposition                                               |
; PURPOSE:  to select a proposition based on local focus constraints.        |
; METHOD:   Sidner's algorithm modified for generation purposes.             |
;           Furthermore, the proposition that is used is marked in the       |
;           as a discourse context so it won't be repeated.                  |
;-----------------------------------------------------------------------------

(define (select-proposition choices PF CF FF)
;--------------------------------------------
 (let* ((focus-preference (local-focus-preference PF CF FF))
        (selection+focus (pick-proposition choices choices focus-preference)))

  (track  *tracing* 1 '(LOCAL FOCUS PREFERENCE ==> ,focus-preference))
  (track  *tracing* 1 '(PREDICATE SELECTED ==> ))
  (trackf *tracing* 1 '(pp-form ',(head selection+focus)) 2)

  (mark-as-used
    (first (head (tail selection+focus)))       ; mark topic used and
    (first (head selection+focus)))             ;  for what discourse purpose

  selection+focus                               ; return selected rp + focus
 )
)

;-----------------------------------------------------------------------------
; FUNCTION: pick-proposition                                                 |
; PURPOSE:  To choose a rhetorical proposition from among several choices    |
;           which will have the same focus as the desired focus and is new   |
;           information (ie has not been uttered already).                   |
; METHOD:   Try to match all the rhetorical propositions with the first     |
;           item in the focus preference list.  Take the first rp that       |
;           succeeds.  If none work, then recurse down the focus-preference  |
;           list until success, otherwise fail.                              |
;           An rp succeeds if the next focus prefered is both within the     |
;           realm of that predicate as well as if that predicate has not     |
;           been used before for that topic (ie only choose to say           |
;           something if you haven't already said it.)                       |
;                                                                            |
; LINGUISTIC PRINCIPLE:                                                      |
;            Select what to say next based (in order of importance) on:      |
;            -- what you want to say                                         |
;            -- what you know                                                |
;            -- what you have already said                                   |
;-----------------------------------------------------------------------------

(define (pick-proposition all-rps rps focus-preference)
;------------------------------------------------------
 (cond
  ((null focus-preference) nil)                 ; no more focus possibilities
  ((null rps)                                   ; no more rhetorical preds
   (pick-proposition                            ;  try next potential focus
     all-rps                                    ;  on all the predicates
     all-rps
     (tail focus-preference))
  )
  ((and
    (not (uttered (head rps) (head focus-preference))) ; pred not uttered on topic
    (member (head focus-preference)             ; next potential focus
            (default-foci (head rps)))          ; member of next pred default foci?
   )
```

```
        (list (head rps)                              ; return rp +
              (list (head focus-preference))          ;  (selected focus
              (alternate-foci (head rps)))            ;   + FF)
       )
       ((and
         (not (uttered (head rps) (head focus-preference))) ; pred not uttered on topic
         (member (head focus-preference)              ; next potential focus
                 (alternate-foci (head rps)))         ; member of next pred alt foci?
        )
        (list (head rps)                              ; return rp +
              (list (head focus-preference))          ;  (selected focus + FF)
              (default-foci (head rps)))
       )
       (t (pick-proposition                            ; otherwise check rest
            all-rps                                    ;  of preds for being in
            (tail rps)                                 ;  focus pref list
            focus-preference)
       )
      )
     )

     ;-------------------------------------------------------------------------------
     ; FUNCTION: uttered                                                            |
     ; PURPOSE:  to determine if the rp has alrealdy been uttered on the given      |
     ;           topic yet by testing context.                                      |
     ;-------------------------------------------------------------------------------

     (define (uttered rp topic)
     ;-------------------------
      (cond ((member (first rp) (context topic))))  ; pred type not uttered yet
     )                                              ;  for this topic

     ;-------------------------------------------------------------------------------
     ; FUNCTION:  any-member                                                        |
     ; PURPOSE:   to determine if any of the items in a list are members of the     |
     ;            membership list.                                                  |
     ;-------------------------------------------------------------------------------

     (define (any-member items membership)
     ;------------------------------------
      (cond
       ((null membership) nil)
       ((member (first items) membership))
       (t (any-member (tail items) membership))
      )
     )

     ;-------------------------------------------------------------------------------
     ; FUNCTION:  local-focus-preference                                            |
     ; PURPOSE:   to determine a focus preference list based on the past foci,      |
     ;            the current focus, and the possible future foci.                  |
     ; LINGUISTIC                                                                   |
     ; MOTIVATION: Apparently people focus on entities just introduced, or, as     |
     ;             in the case of multiple foci, on the related entities in turn.|
     ;-------------------------------------------------------------------------------

     (define (local-focus-preference PF CF FF)
     ;----------------------------------------
      (delete-duplicates                    ; throw away duplicates (just extra work)
       (cond                                ; if
        ((tail CF)                          ;  multiple items in CF (i.e. multiple foci)
         (append                            ;  join lists in preference
          CF
          FF                                ;  the FF of current utterance
          (apply 'append PF)                ;  past foci joined together
         )
        )
        (t
         (append                            ; join lists in preference order:
           FF                               ;  the FF of current utterance
           CF                               ;  CF list
           (apply 'append PF)               ;  past foci joined together
         )
        )
       )
      )
     )

     ;-------------------------------------------------------------------------------
     ; FUNCTION: default-foci and alternate-foci                                    |
     ; PURPOSE:  to return the default-foci (usually translates to agent) or the   |
     ;           alternate-foci (usually translates to patient) of the predicate.|
     ;-------------------------------------------------------------------------------

     (define (default-foci rp) (first (foci rp)))
     (define (alternate-foci rp) (second (foci rp)))
```

```
;-------------------------------------------------------------------------------
; FUNCTION: foci                                                               |
; PURPOSE:  to return the potential foci of a given rhetorical predicate.      |
;           The specific location of focus is rhetorical dependent.            |
; OUTPUT:   ( < current-foci > < potential-foci > )                            |
; LINGUISTIC                                                                   |
; PRINCIPLE: A perlocutionary act generally has a default foci as well as      |
;            alternative foci.  This will be based on the type of act.         |
;-------------------------------------------------------------------------------

(define (foci rp)
;----------------
 (let ((type (first rp)))
  (cond
   ((member type '(attributive))
    (list
     (mapcar 'cadr (second rp)))                ; foci from agent        CF
    )
   ((member type '(definition evidence))
    (cond
     ((tail (first (second rp)))                ; is this an act or result?
      (list
       (mapcar 'cadr (second rp))               ; foci from agent        CF
       (mapcar 'car (third rp))))               ; foci from patient      FF
     (t
      (list
       (mapcar 'car (second rp))                ; foci from agent        CF
       (mapcar 'car (third rp))))               ; foci from patient      FF
     )
    )
   ((member type '(inference))
    (list
     (mapcar 'cadr (second rp))                 ; foci from agent        CF
     (mapcar 'car (third rp)))                  ; foci from patient      FF
    )
   ((member type '(constituent))
    (list
     (mapcar 'car (second rp))                  ; foci from agent        CF
     (append
      (mapcar 'car (third rp))                  ; foci from patient      FF
      (mapcar 'cadr (fifth rp))))               ; foci from beneficiary FF
    )
   ((member type '(compare-contrast))
    (list
     (mapcar 'car (second rp))                  ; foci from agent        CF
     (mapcar 'car (third rp)))                  ; foci from patient      FF
    )
   ((member type '(illustration))
    (list
     (mapcar 'cadr (second rp))                 ; foci from agent        CF
     (mapcar 'car (third rp)))                  ; foci from patient      FF
    )
   ((member type '(cause-effect))
    (list
     (mapcar 'cadr (second rp))                 ; foci from agent1       CF
     (append
      (mapcar 'cadr (fifth rp))                 ; foci from agent2       FF
      (mapcar 'car (third rp))                  ; foci from patient1     FF
      (mapcar 'car (sixth rp))))                ; foci from patient2     FF
    )
   )
  )
 )


(define (sef lyst)
 (cond
  ((null (second lyst)) (first lyst))       ; if no second element, then first
  (t (second lyst))                         ; else return the second element
 )
)
```

```
;////////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                               |
;  MODULE:   ANAPHORA                                                           |
;  PURPOSE:  To perform pronominalization when required by focus mechanism      |
;            for discourse fluidity.                                            |
;  OWNER:    copywrite Mark T. Maybury, June, 1987.                             |
;                                                                               |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////////

;-------------------------------------------------------------------------------
; FUNCTION: use-anaphora?                                                       |
; PURPOSE:  to decide based on the past focus of attention and the current      |
;           focus of attention whether or not to use anaphora.                  |
; LINGUISTIC                                                                    |
; PRINIPLE: If you have just spoken about something (focus on it) and you       |
;           are still speaking about it in the next utterance, you are          |
;           allowed to use refering devices like anaphora.                      |
;-------------------------------------------------------------------------------

(define (use-anaphora? agent focus)
;----------------------------------
 (cond
  ((entity? (second agent))                          ; head noun entity?
   (cond                                             ;  then try to pronom
    ((forefronted? (second agent) focus) t)          ;   forefronted? -> pronom
    ((forefronted? (first agent) focus) t)           ;   forefronted? -> pronom
    (t nil)                                          ;    else focus has shifted,
   )                                                 ;     don't pronominalize
  )                                                  ; else test noun modifier
  ((forefronted? (first agent) focus) t)             ; forefronted? -> pronom
  (t nil)                                            ; else focus has shifted,
 )                                                   ;  don't pronominalize
)

;-------------------------------------------------------------------------------
; FUNCTION:  forefronted?                                                       |
; PURPOSE:   to decide if the provided entity is at the forefront of the       |
;            reader's mind at this point in the discourse.                      |
; LINGUISTIC                                                                    |
; PRINCIPLE: Recency of uttering an entity, as well as saliency, places an      |
;            item at the forefront of readers mind.                             |
;            Future:  Animacy could affect attention reader has given to        |
;                     previous entities (c.f. Fillmore, 1977).                  |
;-------------------------------------------------------------------------------

(define (forefronted? dte focus)
;-------------------------------
 (let ((PF (first focus)))
  (cond
   ((and (member dte (head PF))                      ; dte focused in previous utterance?
         (entity? dte))                              ; dte an entity?
    t)                                               ; -> item is forefronted
   (t nil)                                           ; else not
  )
 )
)

;-------------------------------------------------------------------------------
; FUNCTION: entity?                                                             |
; PURPOSE:  to return t if the item is a KB entity, nil otherwise.              |
;-------------------------------------------------------------------------------

(define (entity? item) (cond ((frame-type item))))
;-------------------------------------------------

;-------------------------------------------------------------------------------
; FUNCTION:  anaphorize                                                         |
; PURPOSE:   to pronominalize a noun phrase in the subject or direct object.    |
;-------------------------------------------------------------------------------

(define (anaphorize syntax-tree location)   ; need focused item also??
;-----------------------------------------
 (cond
  ((eq location 'subject)
   (pronominalize-subject syntax-tree))
  ((eq location 'object)
   (pronominalize-object syntax-tree))
 )
)

(define (pronominalize-subject tree)
;-----------------------------------
 (cond
  ((null tree) nil)
```

```
   ((eq (first (head tree)) 'np)
    (cons
      (pronom (tail (head tree)))
      (tail tree))
    )
    (t (cons (head tree)
             (pronominalize-subject tree))
    )
   )
  )

;-------------------------------------------------------------------------------
; FUNCTION: pronom                                                             |
; PURPOSE:  returns proper pronoun given word features                         |
;           (dictionary 'pronoun)                                              |
;            -> (he   (pronoun pers sing3p subj p3))                           |
;           (dictionary 'determiner)                                           |
;            ->  (this (determiner count sing3p indefart notof noneg nonum))) |
;-------------------------------------------------------------------------------

(define (pronom features)          ; features == (proper-noun sing3p male)
;------------------------
 (cond
  ((eq (first features) 'proper-noun)           ; is it a personal noun?
   (let ((pronouns (dictionary 'pronoun)))      ;  get all pronouns
    (select-pronoun                             ;  match
      (list 'pronoun 'pers (second features)    ;   syntax of proper noun
            'subj 'p3 (third features))
      (mapcar 'look-up pronouns))               ;   entries of pronouns
    )
  )
  (t
   (let ((pronouns (dictionary 'pronoun)))
    (select-pronoun                             ;  match
     (list 'pronoun 'pers (third features)      ; syntax of noun
           'subj 'p3 (fourth features))
     (mapcar 'look-up pronouns))                ;   entries of pronouns
    )
   )
  )
 )


;-------------------------------------------------------------------------------
; FUNCTION: select-pronoun                                                     |
; PURPOSE:  to find a lex entry which matches the provided syntax features.    |
;-------------------------------------------------------------------------------

(define (select-pronoun syntax entries)
;--------------------------------------
 (cond
  ((null entries) nil)
  ((match-syntax syntax (tail (head entries))) ; if syntax matches next entry
   (head entries)                              ; take it (first match)
  )
  (t (select-pronoun syntax (tail entries)))   ; else recurse on tail
 )
)

(define (match-syntax syntax a-list)
;-----------------------------------
 (cond
  ((null a-list) nil)
  ((my-unify syntax (syntax (head a-list))) t)
  (t (match-syntax syntax (tail a-list)))
 )
)
```

SECTION 8

TRANSLATE MODULE

```
;////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                           |
;  MODULE:  TRANSLATE                                                       |
;  PURPOSE: To translate a rhetorical predicate into a case structure which |
;           is transformed into a syntax tree by unifying with GPSG         |
;           + features                                                      |
;  OWNER:   copywrite Mark T. Maybury, June, 1987.                          |
;                                                                           |
;  REPRESENTATION:                                                          |
;  Predicate ->   Focus   ->   Case   -> Relations  -> GPSG -> Orthographic |
;                                                                           |
;  FUNCTIONAL ANALYSIS:                                                     |
;  Discourse -> Pragmatic -> Semantic -> Relational -> Syntactic -> Surface |
;                                                                           |
;  LINGUISTIC                                                               |
;  PRINCIPLES:  The following code implements the translation of the        |
;               message formalism -- rhetorical predicates -- onto surface  |
;               form.  This mapping is motivated by recent insights in      |
;               discourse analysis (c.f. Perlmutter, 1980 and Fillmore,     |
;               1977) which suggest the need for a distinct level of        |
;               grammatical representation in terms of relational           |
;               constituents (e.g. subject, object, predicate). This theory |
;               is supported by the success of interpreters exploiting      |
;               relational knowledge (e.g. GUS (Bobrow, 1977)) and the      |
;               insufficiency of some tactical generators (e.g. Mckeown,    |
;               1985) which do not take sufficient account of relational    |
;               ideas. With this in mind, GENNY first semantically          |
;               interprets a rhetorical predicate into a case formalism     |
;               (c.f. Fillmore, 1977, Sparck-Jones and Boguraev, 1987),     |
;                                 |                                         |
;               the builds relational constituents (Perlmutter, 1980), and  |
;               finally constructs a syntax tree with GPSG (Gazdar, 1982).  |
;               Trees are linearized, and and final utterances are produced |
;               by morphological and orthographic routines.                 |
;                                                                           |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////

;---------------------------------------------------------------------------
; FUNCTION: translate                                                      |
; PURPOSE:  to decide on syntactic, semantic, and pragmatic function.      |
;---------------------------------------------------------------------------

(define (translate rp+focus+context)                ; rp == rhetorical-predicate
;-----------------------------------                 ; focus list = PF CF FF
 (let ((rp (first rp+focus+context))                ; context = given
       (focus (second rp+focus+context))
       (context (third rp+focus+context)))

  (track-blank *tracing* 3 3)
  (track  *tracing* 3 '(= = = = = = = = = = = = = = = = = = = = = = = = =) 1)
  (track  *tracing* 3 '(= = = = = = = = RHETORICAL PREDICATE  = = = = = = =))
  (track  *tracing* 3 '(= = = = = = = = = = = = = = = = = = = = = = = = =) 1)
  (trackf *tracing* 3 `(pp-form ',rp) 2)

  (realize                                          ; realize tree
   (assign-syntax-function                          ; build syntax tree (unify w/grammar)
    (assign-relational-function                     ; make syntactic location choices
      rp                                            ; rhetorical predicate
      (assign-semantic-function rp)                 ; case roles
      (assign-pragmatic-function rp focus context); focus, context, reference
    )                                               ; decisions/information
   )
  )
 )
)

;---------------------------------------------------------------------------
; FUNCTION: assign-pragmatic-function                                      |
; PURPOSE:  to perform pramatic function analysis of the rhetorical        |
;           predicate, determining focus and topic issues.                 |
; METHOD:   Focus shifting algorithm based on Sidner's local focus theory. |
;           Anaphora decisions based on focus information.                 |
;---------------------------------------------------------------------------

(define (assign-pragmatic-function rp focus context)
;---------------------------------------------------
 (list
  (second focus)                    ; discourse-topic-entity
  focus                             ; focus
  context                           ; current context
 )
)

;---------------------------------------------------------------------------
```

```
;-------------------------------------------------------------------------------
; FUNCTION: assign-semantic-function                                           |
; PURPOSE:  To perform a semantic function analysis of the sentence            |
;           by partitioning the rhetorical predicate into case roles.          |
;-------------------------------------------------------------------------------

(define (assign-semantic-function rp)
;------------------------------------
 (list
  (rp-action rp)                                    ; action
  (second rp)                                       ; agent   (one or many w/modifiers)
  (third rp)                                        ; patient (one or many w/modifiers)
  (functional-role 'instrument (fourth rp))         ; instrument
  (functional-role 'location   (fourth rp))         ; location
  (functional-role 'function   (fourth rp))         ; function
  (functional-role 'external-location (fourth rp)) ; external location
  (fifth rp)                                        ; beneficiary (verbs w/ indir obj) [or
agent2]
  (sixth rp)                                        ; manner [or patient2]
  (seventh rp)                                      ; time
  (causal-action rp)                                ; cause
; (eighth rp)                                       ; state
 )
)

;-------------------------------------------------------------------------------
; FUNCTION: functional-role                                                    |
; PURPOSE:  to take a role-list (which originates from the distinguishing      |
;           descriptive attributes (DDA) list in the knowledge base) along     |
;           with a desired role, and return the value for that role.           |
;-------------------------------------------------------------------------------

(define (functional-role role role-list)
;---------------------------------------
 (cond
  ((null role-list) nil)                            ; no more left to check
  ((eq (first (head role-list)) role)               ; found proper role?
   (second (head role-list)))                       ;  return value
  (t (functional-role role (tail role-list)))       ; else tail recurse
 )
)

;-------------------------------------------------------------------------------
; FUNCTION: rp-action                                                          |
; PURPOSE:  to determine the action of the given predicate type.               |
;-------------------------------------------------------------------------------

(define (rp-action rp)
;---------------------
 (let ((type (first rp)))
  (cond
   ((eq type 'definition) 'be)
   ((eq type 'attributive) 'have)    ; for passive -> is attributed to
   ((eq type 'constituent) 'contain) ; has or contains / there are
   ((eq type 'evidence) 'indicate)
   ((eq type 'illustration) 'be)
   ((eq type 'cause-effect) 'be)
   ((eq type 'compare-contrast) 'have)
   ((eq type 'inference) 'be)
  )
 )
)

;-------------------------------------------------------------------------------
; FUNCTION: causal-action                                                      |
; PURPOSE:  to determine the causal action of the given predicate, if poss.    |
;-------------------------------------------------------------------------------

(define (causal-action rp)
 (cond
  ((and (eq 'cause-effect (first rp))                   ; cause-effect pred
        (leaf-node? rp))                                ; lower level of tree
   'indicate)                                           ;       use indicate
  (t nil)
 )
)

;-------------------------------------------------------------------------------
; FUNCTION: leaf-node?                                                         |
; PURPOSE:  to determine the given rp represents entities at leaf node.        |
;-------------------------------------------------------------------------------

(define (leaf-node? rp)
 (cond ((or (act? (cadar (fifth rp))) (result? (cadar (fifth rp))))))
)                                                       ; lower level tree
          ; unmotivated -- make this domain independent in future
```

```
;-------------------------------------------------------------------------------
; FUNCTION: determine-relational-function                                      |
; PURPOSE:  to determine constituent relational information based on           |
;           rhetorical predicate along with focus information and cases.       |
; METHOD:   Active/passive surface form selected in order to emphasize         |
;           focus.  If cf is one of agents then active, else passive.          |
;           Surface insertions are motivated by the rhetorical primitives.     |
;           In the case of the action "be", if the form is also passive,       |
;           then you achieve passivization by means of there insertion.        |
;-------------------------------------------------------------------------------

(define (determine-relational-function rp action agent cf)
;---------------------------------------------------------
 (let ((type (first rp))                                        ; predicate type
       (voice                                                   ; active voice if
        (cond ((equal cf (head (foci rp))) 'active)             ;  cf is subject foci
              ((member action '(be have)) 'active)              ;   can't passivize
              (t 'passive)))                                    ;   else passive
       )
  (cond
   ((and (eq voice 'passive) (eq action 'be)) '(active))  ;  there-insertion
   (t
    (cons
     voice
     (cond
      ((eq type 'illustration) '(example-insertion))
      ((eq type 'inference) '(therefore-insertion))
      ((eq type 'constituent) '(colon-insertion))
      ((eq type 'cause-effect) '(because-insertion))      ; complex sentence?
      (t nil)
     )
    )
   )
  )
 )
)

;-------------------------------------------------------------------------------
; FUNCTION: insertions                                                         |
; PURPOSE:  to insert relational forms (such as "for example") which are       |
;           motivated by the rhetorical predicates.                            |
;                                                                              |
; LINGUISTIC                                                                   |
; MOTIVATION:  Well-connected text is aided by lexical connectives which       |
;              help to indicate the rhetorical role the utterance plays in     |
;              discourse.  The connectives presented here are a subset of      |
;              the taxonomy of markers discussed in Halliday [1985, p 302-7]   |
;-------------------------------------------------------------------------------

(define (insertions relational-structure form)
;---------------------------------------------
 (cond
  ((eq form 'example-insertion)
   (append (list (head relational-structure))
           (list (mapcar 'look-up '(comma for example comma)))
           (tail relational-structure)))
  ((eq form 'colon-insertion)                       ; The brain has two hem: l and r.
   (list (first relational-structure)               ;  subject/agent1
         (second relational-structure)              ;  predicate
         (third relational-structure)               ;  dir-object/patient1
         (fourth relational-structure)              ;  ind-object
         (fifth relational-structure)               ;  ind-object2
         (sixth relational-structure)               ;  ind-object3
         (list (look-up 'colon))                    ;  punctuation
         (seventh relational-structure)             ;  ind-object4/agent2
         (eighth relational-structure)              ;  modifiers/patient2
   )
  )
  ((eq form 'because-insertion)                     ; form complex sentence:
   (list (first relational-structure)               ;  subject/agent1
         (second relational-structure)              ;  predicate
         (third relational-structure)               ;  dir-object/patient1
         (fourth relational-structure)              ;  ind-object
         (fifth relational-structure)               ;  ind-object2
         (sixth relational-structure)               ;  ind-object3
         (list (look-up 'because))                  ;  cause-effect connective
         (seventh relational-structure)             ;  ind-object4/agent2
         (eighth relational-structure)              ;  ind-object5/agent2
         (cond
          ((tenth relational-structure))            ;  predicate2
          (t (second relational-structure)))        ;  repeat predicate
         (ninth relational-structure)               ;  modifiers/patient2
   )
  )
  ((eq form 'therefore-insertion)
   (append (list (head relational-structure))       ; subject/agent1
```

```
                (list (mapcar 'look-up
                              '(comma therefore comma)))      ; therefore connective
                (tail relational-structure)))                 ; rest of relational structure
          ((eq form 'there-insertion)                         ; There are two hem in the brain.
           (cons (look-up 'there) relational-structure))
          ((eq form 'it-extraposition)                        ; It is Alzheimer's disease
           (append (list (list (look-up 'it)))                ;  that the patient has.
                   (list (list (look-up 'be)))                ; left-cleft/topicalisation
                   relational-structure
           )
          )
          (t relational-structure)
         )
        )

        ;-------------------------------------------------------------------------
        ; FUNCTION: assign-relational-function                                   |
        ; PURPOSE:  To take function and form of sentence and generate a structure. |
        ; LINGUISTIC                                                             |
        ; PRINCIPLE:  Motivated by Relational Grammar approach, attempt to embody |
        ;             general, syntax-independent rules which govern relational  |
        ;             constituents (eg subject, object, predicate) rather than   |
        ;             syntactic components (eg noun phrases, verb phrases).      |
        ;-------------------------------------------------------------------------

        (define (assign-relational-function rp case pragmatics)
        ;------------------------------------------------------

         (let* ((action             (first case))          ; Case Roles
                (agent              (second case))
                (patient            (third case))
                (instrument         (fourth case))
                (location           (fifth case))
                (function           (sixth case))
                (external-location  (seventh case))
                (beneficiary        (eighth case))         ; beneficiary
                (manner             (ninth case))          ; (for verbs w/indir object)
                (time               (tenth case))
                (causal-action      (tenth (tail case)))   ; for complex sentences
        ;       (state              (eleventh case)))      ;  as in Q is R since X indicates Y

                (topic              (first pragmatics))    ; discourse-topic-entity
                (focus              (second pragmatics))   ; FF-CF-PF
                (context            (third pragmatics))    ; previously uttered entities

                (relations (determine-relational-function rp action agent (second focus)))
                (voice (first relations))          ; voice == active or passive
                (form  (second relations))         ; form  == for-example, it-extraposition

                (tense (cond (time) (t 'pres)))    ; tense == pres (default) or past

                (predicate   (make-v  action tense voice))         ; relational constituents
                (subject     (make-np agent focus context 'and))   ; list of agents/patients
                (dir-object  (make-np patient focus context 'and)) ; make all nps + conj
                (ind-object  (make-pp instrument 'with focus context))
                (ind-object2 (make-pp function   'for focus context))
                (ind-object3 (make-pp location   'located-in focus context))
                (ind-object4 (make-pp external-location 'on focus context))
                (ind-object5 (make-np beneficiary focus context 'and))
                (modifiers   (make-np manner focus context 'and))
                (predicate2  (make-v  causal-action tense voice)) ; second verb

               )

        ;_________________________________________________________________________;
        ;        dynamic tracking routines for monitoring program behavior        ;
        ;                                                                         ;
         (track  *tracing* 3 '(PRAGMATIC FUNCTION [discourse-topic-entity/focus/given]: ))
         (trackf *tracing* 3 '(pp-form ',pragmatics) 3)                          ;
        ;                                                                        ;
         (track  *tracing* 3 '(SEMANTIC FUNCTION :))                             ;
         (track  *tracing* 3 '(action agent patient inst loc funct manner time cause))
         (trackf *tracing* 3 '(pp-form ',case) 3)                                ;
        ;                                                                        ;
         (track  *tracing* 3 '(RELATIONAL FUNCTION [voice and form': ,relations) 2)  ;
        ;                                                                        ;
        ;________________________________________________________________________;

          (cond
           ((or (null subject) (null dir-object)) nil)                  ; no full sentence
           ((eq voice 'active)                                          ; active voice
            (let ((final-order
                   (apply 'append
                    (insertions (list subject predicate dir-object ind-object
                                      ind-object2 ind-object3 ind-object4
                                      ind-object5 modifiers predicate2)
```

```
                    form))))
                (cond ((null final-order) nil)                          ; save final order
                      (t (list final-order)))
               )
              )
              ((eq voice 'passive)                                      ; passive voice
               (let ((final-order                                       ;  promote dir-obj
                      (apply 'append                                    ;  to subject position
                       (insertions (list dir-object predicate subject ind-object
                                         ind-object2 ind-object3 ind-object4 ind-object5 modifiers)
                        form))))
                (cond ((null final-order) nil)                          ; save final order
                      (t (list final-order)))
               )
              )
             ) ; cond
            ) ; let
           ) ; fnc


;-------------------------------------------------------------------------------
; FUNCTION: rem-nil                                                            |
; PURPOSE:  to remove nil's from a list but preserve list structure.           |
;-------------------------------------------------------------------------------

(define (rem-nil l)
;-------------------
 (cond
  ((null l) nil)                              ; stop if finished
  ((null (head l)) (rem-nil (tail l)))        ; next null? -> drop it
  (t (cons (head l) (rem-nil (tail l))))      ; else keep it and recurse
 )
)

;-------------------------------------------------------------------------------
; FUNCTION: assign-syntax-function                                             |
; PURPOSE:  To unify a lexical list with the given grammar and produce a       |
;           consistent and well-featured syntax tree.                          |
; METHOD:   Utilises a chart data structure which acts as a well-formed        |
;           substring table containing all the possible structures.            |
;           First the routine generate-tree generates the chart.               |
;           At the end of the unification process, the tree generation         |
;           procedure returns a list of possible edges which represent         |
;           all the possible ways to realize the given input.  The             |
;           procedure output-parses-features retrieves the successful          |
;           realizations from the chart by consistently percolating            |
;           features up the edges which constitute the tree.                   |
;           As in interpretation, there may be more than one possible way      |
;           to realize a given structure.                                      |
;-------------------------------------------------------------------------------

(define (assign-syntax-function lexical-list)
;--------------------------------------------
 (let ((lexical-list (first lexical-list)))

  (track  *tracing* 3 '(LEXICAL INPUT TO SENTENCE GENERATOR:))
  (trackf *tracing* 3 '(pp-form ',lexical-list) 2)

  (generate-tree lexical-list 's *grammar*)     ;  realizes surface form

  (track  *tracing* 3 '(SYNTAX OUTPUT FROM SENTENCE GENERATOR:))
  (trackf *tracing* 3 '(output-parses-features
                        (find-feature-parses 's ',*chart*)) 2)

  (first                                        ;  select the first successful
   (parse-tree-feature-list                     ;   realization for now
    (find-feature-parses 's *chart*)))
 )
)
```

RELATIONAL GRAMMAR MODULE

```
;////////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                               |
;  MODULE:    RELATIONAL GRAMMAR                                                |
;  PURPOSE:   To make decisions on the relations and determining of             |
;             constituents given semantic (ie role) information as well         |
;             as pragmatic information.                                         |
;  OWNER:     copywrite Mark T. Maybury, June, 1987.                            |
;                                                                               |
;  LINGUISTIC                                                                   |
;  PRINCIPLES:  This code is motivated by relational grammar (Perlmutter,       |
;               1980), which suggests interpretation and generation requires    |
;               a distinct level of representation between syntax and           |
;               semantics.  Furthermore, this theory shows some promise         |
;               in language portability.  The syntactic experts which           |
;               build relational constituents are constrained by pragmatic      |
;               (e.g. given/new context and focus) as well as syntactic         |
;               knowledge.  While these syntactic experts are language          |
;               dependent, and would therefore need to be rebuilt for a new     |
;               target language, the relational constituents which they         |
;               represent have shown promise in universality (c.f. Cole         |
;               and Sadock, 1977).                                              |
;                                                                               |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////////


;-------------------------------------------------------------------------------
; FUNCTION: make-np                                                             |
; PURPOSE:  to generate a noun phrase based on the agent(s) given               |
; METHOD:   Select lexical entry                                                |
;           Generate NP                                                         |
; INCOMING: ((brain)) -> (the brain)                                            |
;           ((lobe lfrontal) (test language)) -> the lfrontal lobe & the        |
;           ((lobe) (test))  -> the lobe and the test                           |
;           ((value test indef 1)) -> a damage value of one                     |
;           ((hemisphere two)) -> two hemispheres                               |
;-------------------------------------------------------------------------------

(define (make-np agents focus context connective)
;------------------------------------------------
 (conjunction
  (make-all-nps agents focus context)
  connective)
)

(define (make-all-nps agents focus context)
;------------------------------------------
 (cond
  ((null agents) nil)
  (t
   (append
    (list (make-nps (head agents) focus context))       ; make first np
    (make-all-nps (tail agents) focus context)          ; tail recurse
   )
  )
 )
)

(define (lister item) (cond ((null item) nil) (t (list item))))
;--------------------

;-------------------------------------------------------------------------------
; FUNCTION: make-nps                                                            |
; PURPOSE:  to generate a noun phrase constrained by:                           |
;            - pragmatic knowledge of local focus and context (eg referents)    |
;            - relational knowledge of phrasal components                       |
;            - morphological knowledge of lexical agreement (eg a, an)          |
;                                                                               |
; METHOD:   If previous discourse topic entity (eg focus) is continuing,        |
;              then replace np with appropriate anaphora.                       |
;           Else build an appropriate noun phrase.                              |
;                                                                               |
; LINGUISTIC                                                                    |
; STRUCTURE: NP == quantifier article modifiers head post-modifiers             |
;                                                                               |
;-------------------------------------------------------------------------------

(define (make-nps agent focus context)
;-------------------------------------
; agent == (head + modifier + determiner quantifier)
 (cond
  ((use-anaphora? agent focus)                          ; if forefronting in reader
   (let* (descr (get-entry (look-up (first agent))))    ; use anaphoric reference
     (list (pronom (syntax (get-entry (look-up (first agent))))))))
   )
```

```
                )
                (t
                 (let* ((head-entries (lister (look-up (first agent)))) ; check head
                        (head       (get-entry (car head-entries)))     ; get lexical cat of some entry
                        (modifiers (lister (look-up (second agent))))   ; look modifier (adj/ordinal/nom
inal)
                        (modifier2 (lister (look-up (fifth agent))))    ; modifer2
                        (after-det (cond ((get-entry (car modifier2)))  ; determine word
                                         ((get-entry (car modifiers))) ; following determiner
                                         (t head)))                    ; for morph agreement
                        (determiner (select-determiner agent head context)); select determiner
                        (article (cond ((lister (choose-article after-det determiner))))) ; default to
                        (of-num  (cond ((fourth agent)
                                        (list (look-up 'of) (look-up (fourth agent))))))
                        (quantifier (lister (sixth agent)))
                       )
                  (append quantifier article modifier2 modifiers head-entries of-num)
                 )
                )
               )
              )


              ;-------------------------------------------------------------------------
              ; FUNCTION: head-type, head-syntax                                       |
              ; PURPOSE:  to determine the category or syntax of the first lexical entry. |
              ;-------------------------------------------------------------------------

              (define (head-type entries) (word-type (first (tail entries))))
              ;--------------------------------------------------------------

              (define (get-entry entries) (first (tail entries)))
              ;---------------------------------------------------

              ;-------------------------------------------------------------------------
              ; FUNCTION: choose-article                                               |
              ; PURPOSE:  to select an appropriate article from the dictionary given the |
              ;           linear lexical successor of the determiner.                  |
              ;-------------------------------------------------------------------------

              (define (choose-article entry type)
              ;-----------------------------------
               (cond
                ((eq type nil) nil)                                 ; if none
                ((eq type 'indef)                                   ; if indef art check
                 (look-up (indefinite (realization entry))))       ;  spelling of np head
                (t (look-up (indefinite (realization entry))))
              ; (t (look-up 'the))                                  ; default to indef article
               )
              )


              ;-------------------------------------------------------------------------
              ; FUNCTION: indefinite                                                   |
              ; PURPOSE:  to determine the proper indefinite article based on the first |
              ;           character of the provided word.                              |
              ;-------------------------------------------------------------------------

              (define (indefinite word)
              ;-------------------------
               (cond ((member (first (explode word)) '(a e i o u)) 'an) (t 'a))
              )

              ;-------------------------------------------------------------------------
              ; FUNCTION:  select-determiner                                           |
              ; PURPOSE:   to select a determiner for the np guided by pragmatic       |
              ;            constraints.                                                |
              ; THEORY:    New information is generally introduced by the indefinite art. |
              ;            Given information is generally introduced by definite article. |
              ;            NB: new information is that which has not yet been uttered to |
              ;            the speaker/reader in the current discourse.  Given information|
              ;            is all that has been generated for the speaker/reader.      |
              ;            If mass or proper-noun, no article.                         |
              ;            If compound noun, complex noun (eg hyphenated) or modifier->def|
              ;                                                                        |
              ;------------------------------------------------------ ----------------

              (define (select-determiner np-skeleton entry context)
              ;-----------------------------------------------------
               (let ((category (word-type entry)))
                (cond
                 ((eq category 'proper-noun) nil)                   ; if proper noun no article
                 ((eq category 'pronoun) nil)                       ; if pronoun no article
                 ((eq (noun-type entry) 'mass) nil)                 ; if mass noun no article
                 ((eq (third np-skeleton) 'none) nil)               ; suppressed article?
                 ((eq (word-type (get-entry (look-up
```

```
            (second np-skeleton)))) 'proper-noun) nil); modifier proper-noun?
       ((eq (third np-skeleton) 'indef) 'indef)      ; indef article?
       ((eq (third np-skeleton) 'def) 'def)          ; suppressed article?
       ((given? (first np-skeleton) context) 'def)   ; information given? -> def
       ((not (null (second np-skeleton))) 'def)      ; modifier to restrict interp? -> def
       ((not (null (fifth np-skeleton))) 'def)       ; compound noun? -> def
       ((complex-noun? (realization entry)) 'def)    ; complex-noun?  -> def
;      ((complex-noun? (first np-skeleton)) 'def)    ; complex-noun? -> def
       ((new? (first np-skeleton) context) 'indef)   ; information new? -> indef
       (t 'indef)                                    ; else default to indef
      )
     )
    )

    ;-------------------------------------------------------------------------
    ; FUNCTION: given?                                                       |
    ; PURPOSE:  to determine if an entity has been mentioned previously (in  |
    ;           recent conversation) and, therefore, is known or in the "front |
    ;           of the mind" of the reader/listener.                         |
    ;-------------------------------------------------------------------------

    (define (given? entity context)
    ;------------------------------
     (member entity context))              ; simply past focus is not rich enough

    ;-------------------------------------------------------------------------
    ; FUNCTION: new?                                                         |
    ; PURPOSE:  to determine if an entity can be considered as new information |
    ;           by testing if it is just being mentioned in context for the  |
    ;           first time.                                                  |
    ;-------------------------------------------------------------------------

    (define (new? entity context)
    ;----------------------------
     (not (member entity context)))                        ; not already utterred

    ;-------------------------------------------------------------------------
    ; FUNCTION: complex-noun?                                                |
    ; PURPOSE:  to determine if the provided entity is a complex noun.       |
    ;-------------------------------------------------------------------------

    (define (complex-noun? noun)
    ;---------------------------
     (member '- (explode noun)))           ; if has a hyphen, it's a complex noun

    ;-------------------------------------------------------------------------
    ; FUNCTION: conjunction                                                  |
    ; PURPOSE:  to take a list of constituents and return a list with the    |
    ;           linear order of constituents, appropriate punctuation, and   |
    ;           insertion of the connective.                                 |
    ;-------------------------------------------------------------------------

    (define (conjunction constituents connective)
    ;--------------------------------------------
     (cond
      ((null (tail constituents)) (head constituents)) ; only one?
      ((null (tail (tail constituents)))            ; just before end?
       (append
         (head constituents)                          ; take next to last
         (list (look-up connective))                  ;      connective
         (first (tail constituents)))                 ;      last item
      )
      (t (append
           (head constituents)                        ; else tail recurse
           (list (look-up 'comma))                    ;  building list with punct.
           (conjunction  (tail constituents) connective))
      )
     )
    )


    ;-------------------------------------------------------------------------
    ; FUNCTION: make-v                                                       |
    ; PURPOSE:  to generate a verb structure based on the action and tense.  |
    ; METHOD:   Select lexical realizations of action                        |
    ;           Select rules based on entry syntax                           |
    ;           Choose article if necessary                                  |
    ;           Choose particle if necessary for verbal phrase               |
    ;           Generate NP                                                  |
    ;-------------------------------------------------------------------------

    (define (make-v action tense voice)
    ;----------------------------------
     (let* ((entries (look-up-verb action))       ; search lexicon for action realizations
            (head (head-type entries))            ; use for aux selection in future
            (auxiliary                            ; select an appropriate aux
```

```
                (choose-aux action tense voice)))
        (cond
         (auxiliary
          (append                                   ; if require auxiliary
            auxiliary                               ;  aux
            (list (list action                      ;  + verb
                        (make-verbs-past-participle ;  tense = past particple
                          (a-list action))))
            (verbal-phrase action voice)            ;  + particle for verbal phrase
          )
         )
         (entries (list entries))                   ; else just entries if any
         (t nil)
        )
       )
      )

      ;-------------------------------------------------------------------------
      ;                                                                         |
      ;  FUNCTION: make-verbs-past-participle   ("eat" -> "eaten")              |
      ;  PURPOSE:  To return the list of dictionary descriptions which appear in |
      ;            the given a-list of some word, filtered so that the verbs are |
      ;            past-participle.                                             |
      ;  INPUT:    the association list for the word                            |
      ;  OUTPUT:   The modified a-list                                          |
      ;                                                                         |
      ;-------------------------------------------------------------------------

      (define (make-verbs-past-participle a-list)
      ;-------------------------------------------
       (cond
        ((null a-list) nil)
        ((eq (word-type (head a-list)) 'verb)               ; change only if verb
         (append                                            ;  attach fixed head to tail
          (list                                             ;  fix head
           (cons
              (append1
                 (my-delete 'verb
                  (subst 'past 'pres (syntax (head a-list))))); change to past-participle
                 'en)
              (list (semantics (head a-list)))              ; save semantics
              (list (realization (head a-list)))            ; save realization
           )
          )
          (make-verbs-past-participle (tail a-list))        ;  check tail
         )
        )
        (t (make-verbs-past-participle (tail a-list)))      ; else check only tail
       )                                                    ; (throws out non verbs)
      )



      ;-------------------------------------------------------------------------
      ; FUNCTION: choose-aux                                                    |
      ; PURPOSE:  to select an appropriate auxiliary from the dictionary given  |
      ;           the word entry.                                               |
      ;-------------------------------------------------------------------------

      (define (choose-aux action tense voice)
      ;---------------------------------------
       (cond
        ((eq voice 'passive)                            ; if voice passive
         (cond                                          ;  non-passifiable verb?
          ((member action '(be have)) nil)              ;   return nil
          (t (list (look-up-verb 'be)))))               ;  else return auxiliary
        (t nil)                                         ; otherwise there is none
       )
      )

      ;-------------------------------------------------------------------------
      ; FUNCTION: verbal-phrase                                                 |
      ; PURPOSE:  to return the appropriate verbal particle to complete the     |
      ;           verbal phrase in the passive tense.                           |
      ;-------------------------------------------------------------------------

      (define (verbal-phrase verb voice)
      ;----------------------------------
       (cond
        ((eq voice 'passive)                                ; passive voice?
         (cond                                              ; return particle based
          ((eq verb 'contain)  (list (look-up 'in)))        ;  upon verb type
          ((eq verb 'indicate) (list (look-up 'by)))
          (t nil)                                           ;  or nothing at all
         )
        )
```

```
      (t nil)                                                    ; otherwise nil
     )
    )

    ;-------------------------------------------------------------------------------
    ; FUNCTION: make-pp                                                            |
    ; PURPOSE:  to generate a pp structure based on the object provided.           |
    ; METHOD:   Select lexical entry                                               |
    ;           Select rules based on entry syntax                                 |
    ;           Choose article if necessary                                        |
    ;           Generate pp                                                        |
    ;-------------------------------------------------------------------------------

    (define (make-pp object relation focus context)
    ;-----------------------------------------
     (let* ((entries (look-up (first object)))    ; search dictionary
            (preposition (choose-prep relation))) ; select an appropriate preposition

      (cond
       ((null entries) nil)                       ; no realization or no object
       ((null preposition) nil)                   ; no preposition -> nil
       (t (append preposition (make-nps object focus context)), ; else build true np
      )
     )
    )


    ;-------------------------------------------------------------------------------
    ; FUNCTION: choose-preposition                                                 |
    ; PURPOSE:  to select an appropriate preposition from the dictionary given     |
    ;           word entry.                                                        |
    ;-------------------------------------------------------------------------------

    (define (choose-prep type)
    ;-------------------------
     (cond
      ((eq type 'located-in) (list (look-up 'located) (look-up 'in)))
      (t (list (look-up type)))
     )
    )

    ;-------------------------------------------------------------------------------
    ; FUNCTION: select-word                                                        |
    ; PURPOSE:  to choose a word from a list given a constraining syntax           |
    ;           and return a word description with a bound feature list.           |
    ;-------------------------------------------------------------------------------

    (define (select-word syntax word-list)
    ;--------------------------------------
     (let* ((variable-counter 30)
            (features (rechristen-variables syntax))
            (word-syntax (rechristen-variables (syntax (head word-list)))))
      (cond
       ((my-unify features word-syntax)             ; does this word unify?
        (list                                       ; if yes, return it
          (my-bind features                         ; instantiated
                   (my-unify features word-syntax))
          (semantics (head word-list))
          (realization (head word-list))))
       (t (select-word syntax (tail word-list))))
     )
    )
    ;-------------------------------------------------------------------------------
    ; FUNCTION: get-feature                                                        |
    ; PURPOSE:  to return the feature/value list given the feature name & syntax   |
    ;-------------------------------------------------------------------------------

    (define (get-feature feature syntax)
    ;-----------------------------------
     (cond
      ((null syntax) nil)
      ((listp (head syntax))
       (cond ((eq feature (first (head syntax))))))
      (t (get-feature feature (tail syntax))))
    )
```

SECTION 10

GENERATE MODULE

```
;////////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                               |
;  MODULE:    GENERATE COMPILE                                                  |
;  PURPOSE:   To compile routines which perform syntactic generation.           |
;  OWNER:     copywrite Mark T. Maybury, June, 1987.                            |
;  LINGUISTIC                                                                   |
;  PRINCIPLE: This code loads the syntactic modules which use a unification     |
;             feature grammar together with lexical entries to generate         |
;             a well-formed syntactic tree.  This eventually will be            |
;             linearised, and the syntactic categories will guide               |
;             selection of morphology.                                          |
;                                                                               |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////////


;*** declare global variables, macros compiled and evaled during compile ***


(declare (special *tracing* *grammar* *non-terminals*
                  *syntax-grammar*
                  *grammarloaded* *dictionaryloaded* *knowledgebaseloaded*
                  *sentenceparsed*
                  *agenda* *chart* *stack*
                  *response*)
         (macros t))

(include ~/lisp/semantics/localf.parse)   ; local functions for efficiency

  ;** following used for debugging:

;(include /user/sgp/lispaids/treeprint.l) ; tree printing functions
(include ~/lisp/syntax/find_parses.l)     ; routines for listing rule firings
(include ~/lisp/syntax/rule_trees.l)      ; routines for listing rule firings
(include ~/lisp/syntax/debug.l)           ; debug routines

    ;*** dictionary routines ***

(include ~/lisp/dictionary/dictionary_macros.l) ; functions to create a dictionary
(include ~/lisp/dictionary/makedictionary.l)    ; functions to create a dictionary
(include ~/lisp/dictionary/genlookup.l)         ; functions for dictionary look-up
;(include ~/lisp/dictionary/lookup.l)            ; functions for dictionary look-up
;(include ~/lisp/dictionary/root.l)              ; root finder for words

    ;*** generator and chart routines ***

(include ~/lisp/grammars/gen_cfg2.l)      ; functions for generating sentences from CFG
(include ~/lisp/syntax/category.l)        ; functions for lex category search/match
(include surface_generator.l)             ; syntax tree generation routines
(include compilegram.l)                   ; grammar compile routines
(include preprocess.l)                    ; local routines for preprocessing
(include ~/lisp/syntax/preprocess.l)      ; routines for preprocessing
(include ~/lisp/syntax/vertex_edge.fast)  ; routines for vertices and edges
(include ~/lisp/syntax/unification.l)     ; routines for unification + binding
(include ~/lisp/syntax/feature_parses.l)  ; routines for parsing with features
(include ~/lisp/syntax/feature_process.fast) ; processing feature routines
```

```
;;****************** TREEPRINTER *********************************************
;; prints a labelled bracketing out as a tree. If no second argument
;; is given, output is assumed to be the standard (i.e. terminal).
;; A second argument must evaluate to an output port:
;; (treeprint <tree> <port>)
;; Syntax of input is:

;; <tree> ::=  (<node-label> <daughter>*)
;; <daughter> ::= <tree>
;;
;; terminals are the case where the node label is the only member of the tree
;; e.g.
;;
;; (treeprint '(s (np (n (joe))) (vp (v (likes)) (np (n (mary)
;;
;; Where node-labels are not atoms, the fn 'node-label' will need changing.
;;
;; NB maximum width governed by Franz defaults

;;; ***** ************************MAIN FNS*****************************


;; treeprint is a lexpr:

(defun treeprint (s &optional (port nil))

 (setq s (make-tree-level s (find-max-depth s 0)))
 (setq s (replace-all-nodes s nil))
 (annotate-nodes-with-width-info  s 0 nil)
 (treeprint-aux (list s) port nil)

; first the tree is adjusted so that all the terminals are at the same
; depth from the root. Dummy nodes are used to make the tree a uniform
; object. Both they and the actual nodes are replaced by gensyms, and attached
; to the gensym as a property by replace-all-nodes.
; 'annotate-nodes-with-width-info' goes down the tree
; recursively attaching to each node as a property its width:
; the number of characters in the widest symbol from it to the terminal
; it dominates (or sum of such if a branching node). This is used by the
; actual printing functions to centre each node properly.
;

(def treeprint-aux
 (lambda (s port next)
 (if (setq next (print-nodes s  nil port))
    then (print-downstrokes s port)
         (if (no-branching-in-any s)
            then (treeprint-aux next port nil)
         else (print-horizontals s port)
              (treeprint-aux next port nil))
  else (terpr port)

; ****** LEVELLING A TREE*****************************
; extra dummy nodes of form e.g. '**' are added to make sure all the
; terminals are at the same depth. This makes life easier, and usually
; looks better

(def is-preterminal
 (lambda (i)
  (and (atom (car i))(atom (caadr i))(null (cdadr i)

(def make-tree-level
 (lambda (tree depth)
 (if (is-preterminal tree)
   then (add-extra-depth tree depth)
 else (cons (car tree)
            (mapcar '(lambda(x)(make-tree-level x (sub1 depth)))
                    (cdr tree)

; goes up and down a tree adding extra nodes

(def find-max-depth
 (lambda (i counter)
 (if (null i) then counter
  elseif (atom (car i))
    then (max counter (find-max-depth (cdr i) counter))
  else (max (find-max-depth (car i) (add1 counter))
            (find-max-depth (cdr i) counter)

(def make-dummy-node
 (lambda (node)
  (implode (mapcar '(lambda (x)(quote *)) (explode node)

; makes node consisting of *s, same length as original
```

```
(def add-extra-depth
 (lambda (pre-termn extra-depth)
 (list (car pre-termn)
       (add~extra-depth-aux
         (make-dummy-node (car pre-termn))
         (cadr pre-termn)
         (sub1 extra-depth)]

(def add-extra-depth-aux
 (lambda (dummy terminal extra-depth)
  (if (lessp extra-depth 1) then terminal
   else (list dummy (add-extra-depth-aux dummy terminal (sub1 extra-depth]

; *** LABELLING THE NODES ************************************
; each node in the tree is replaced by a new symbol, acting as an
; index for information about how wide the constituent is, and how big
; the node itself is

(def replace-all-nodes
 (lambda (tree label )
 (if (null (cdr tree))
     then (setq label (gensym (car tree)))
          (putprop label
                (node-label (car tree))
                'category)
          (list label)
  else (setq label (gensym (car tree)))
       (putprop label
                (node-label (car tree))
                'category)
       (cons label (mapcar '(lambda (x)(replace-all-nodes x nil)) (cdr tree]

(def node-label
 (lambda (i)
 (if (eq '* (car (explode i))) then '!
  else i]

; for a grammar using non-atomic node-labels this fn has to be different


;***** **************************** WIDTHS ****************************

(def annotate-nodes-with-width-info
 (lambda (tree biggest-so-far nodelength)
  (setq nodelength (atom-length (get (car tree) 'category)))
  (if (null (cadr tree))  ;terminal
    then (putprop
           (car tree)
           (setq biggest-so-far (add1 (max biggest-so-far nodelength)))
           'width)
          biggest-so-far
  else (putprop (car tree)
           (setq biggest-so-far
               (sum-widths (cdr tree)(max biggest-so-far nodelength)))
            'width)
       biggest-so-far]

(def sum-widths
 (lambda (tree-list biggest-yet)
 (if (null tree-list) then 0
 else (plus (annotate-nodes-with-width-info (car tree-list) biggest-yet nil)
            (sum-widths (cdr tree-list) biggest-yet]

(def atom-length
 (lambda (a)
  (length (explode a]

;****** PRINTING A NODE ****************************

(def print-nodes
 (lambda (old new port)
 (if (null old) then (terpr port)
                     new
 else (print-node (car old) port)
      (print-nodes (cdr old )(append new (cdr (car old))) port]

(def print-node
 (lambda (const port)
   (prog (lngth diff)
   (setq lngth  (atom-length (get (car const) 'category)))
   (setq diff (difference (get (car const) 'width) lngth))
   (msg (P port)(B (quotient diff 2)))
   (print (get (car const) 'category) port)
   (if (evenp diff) then (msg (P port) (B (quotient diff 2)))
   else (msg (P port)(B (add1 (quotient diff 2)))))
```

```
    (return t)

;************************  VERTICAL LINES ******************************

(def print-downstrokes
 (lambda (cl port)
 (if (null cl) then (terpr port)
 else (print-downstroke (car cl) port)
      (print-downstrokes (cdr cl) port)

(def print-downstroke
 (lambda (const port)
   (prog (diff)
   (setq diff (difference (get (car const) 'width) 1))
   (msg (P port)(B (quotient diff 2)))
   (print '! port)
   (if (evenp diff) then (msg (P port)(B (quotient diff 2)))
   else (msg (P port)(B  (add1 (quotient diff 2))))))
   (return t)

; ********************** PRINT HORIZONTALS:*****************************

(def no-branching-in
 (lambda (i)
  (eq (length i) 2)

(def no-branching-in-any
 (lambda (l)
 (if (null l) then t
 elseif (no-branching-in (car l))
      then (no-branching-in-any (cdr l)

(def print-horizontals
 (lambda (cl port)
 (if (null cl) then (terpr port)
  elseif (no-branching-in (car cl))
    then (print-downstroke (car cl) port)
         (print-horizontals (cdr cl) port)
  else (print-hor-aux (cdar cl) t port)
       (print-horizontals (cdr cl) port)

(def dashes
 (lambda (n port)
 (if (zerop n) then nil
  else (print '_ port)
       (dashes (sub1 n) port)

(def print-hor-aux
 (lambda (cdaughters is-first port)
   (prog (diff)
   (setq diff (difference (get (caar cdaughters) 'width) 1))
   (if (null (cdr cdaughters))
     then (dashes (add1 (quotient diff 2)) port)
          (if (evenp diff) then (msg (P port)(B (quotient diff 2)))
           else (msg (P port)(B (add1 (quotient diff 2))) ))
   elseif is-first
     then (msg (P port)(B (quotient diff 2)))
          (dashes 1 port)
          (if (evenp diff) then (dashes (quotient diff 2) port)
           else (dashes (add1 (quotient diff 2)) port) )
          (print-hor-aux (cdr cdaughters) nil port)
  else (dashes (get (caar cdaughters) 'width) port)
       (print-hor-aux (cdr cdaughters) nil port)
```

```
;*****************************************************************************
;                                                                            *
;  FUNCTION: complete-edges                                                  *
;  PURPOSE:  To return the subset of the list of given edges which are       *
;            complete edges.                                                 *
;                                                                            *
;*****************************************************************************

(define (complete-edges edges)
 (cond
  ((null edges) nil)                          ; no more left? -> go home
  ((complete? (head edges))                   ; first complete?
   (cons (head edges)                         ;  save it
         (complete-edges (tail edges))))      ;  check for complete in tail
  (t (complete-edges (tail edges)))           ; else just check tail
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: incomplete-edges                                                *
;  PURPOSE:  To return the subset of the list of given edges which are       *
;            incomplete edges.                                               *
;                                                                            *
;*****************************************************************************

(define (incomplete-edges edges)
;################################
 (cond
  ((null edges) nil)                          ; no more left? -> go home
  ((incomplete? (head edges))                 ; first incomplete?
   (cons (head edges)                         ;  save it
         (incomplete-edges (tail edges))))    ;  check for incomplete in tail
  (t (incomplete-edges (tail edges)))         ; else just check tail
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: output-trees                                                    *
;  PURPOSE:  To print ou. a tree listing of the parsings.                    *
;                                                                            *
;*****************************************************************************

(define (output-trees  successful-edges)
;#######################################
 (let ((trees (parse-tree-list successful-edges)))
  (blank 2)
  (mapc 'treeprint trees)
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: output-parses                                                   *
;  PURPOSE:  To print out a nice listing of the parsings.                    *
;                                                                            *
;*****************************************************************************

(define (output-parses successful-edges)
;#######################################
 (let ((trees (parse-tree-list successful-edges)))
  (blank 2)
  (princ (eval `(pp ,trees)))
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: parse-tree-list                                                 *
;  PURPOSE:  To translate a list of successful edges into parse trees.       *
;                                                                            *
;*****************************************************************************

(define (parse-tree-list successful-edges)
;#######################################
  (mapcar 'make-tree successful-edges)
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: make-tree                                                       *
;  PURPOSE:  To generate a tree parse structure from the information on      *
;            the edge.                                                       *
;                                                                            *
```

```
;*****************************************************************************

(define (make-tree edge)
;########################
 (cond
  ((is-a-category (category-of edge))              ; if edge is lexical
   (list (category-of edge) (contents-of edge))    ;  list (category contents)
  )
  (t (cons
       (category-of edge)                          ; head = cat of edge
       (mapcar 'make-tree (contents-of edge))      ; tail = trees of contents
     )
  )
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: find-parses                                                     *
;  PURPOSE:  To find all the parses from the chart by finding all the        *
;            complete parses starting with edges from start.                 *
;                                                                            *
;*****************************************************************************

(define (find-parses startsymbol chart)
;######################################
 (let ((start  (first chart))                     ; start = first vertex in chart
       (finish (car (last chart))))               ; finish = last vertex in chart
  (find-all-parses
     startsymbol
     finish
     (edges-out start)                            ; examine the edges from start
  )
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: find-all-parses                                                 *
;  PURPOSE:  To recurse on the list of all edges from the start vertex and   *
;            test to see if they meet the three conditions required to be    *
;            a legal parse:                                                  *
;              1 -- edge out must have start symbol label.                   *
;              2 -- the right vertex of edge must be the finish vertex.      *
;              3 -- there must be no more required constituents.             *
;                                                                            *
;*****************************************************************************

(define (find-all-parses startsymbol finish edges-out-of-start)
;##############################################################
 (cond
  ((null edges-out-of-start) nil)                  ; finished -> quit
  ((and                                            ; parse good if:
    (eq (category-of (head edges-out-of-start))    ; 1 edge is start
        startsymbol)                               ;    symbol and
    (eq (right-vertex-of (head edges-out-of-start)) ; 2 right vertex is
        finish)                                    ;    last vertex and
    (eq (needed-of (head edges-out-of-start))      ; 3 no needed constituents
        nil))
   (cons (head edges-out-of-start)                 ; keep head of list with
         (find-all-parses                          ; examine tail
            startsymbol
            finish
            (tail edges-out-of-start))
   )
  )
  (t (find-all-parses                              ;  else examine tail
       startsymbol
       finish
       (tail edges-out-of-start))
  )
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: remove-edge                                                     *
;  PURPOSE:  To get the next edge.  By altering this routine we can vary     *
;            the search process.  If we pop the item from the agenda, we     *
;            will search depth first.  By removing the next edge from the    *
;            bottom of the agenda we will search breadth first.              *
;                                                                            *
;*****************************************************************************

(define (remove-edge agenda)
;###########################
```

```
    (my-pop (eval agenda))
)
```

```
;////////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                               |
;  MODULE:  RULE TREES AND TRACE                                                |
;  PURPOSE: To provide support for retrieving rules applied.                    |
;                                                                               |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////////

;*******************************************************************************
;                                                                              *
;  FUNCTION: output-rule-trees                                                 *
;  PURPOSE:  To print out a tree listing of the rules applied.                 *
;                                                                              *
;*******************************************************************************

(define (output-rule-trees successful-edges)
;############################################
 (let ((trees (rule-tree-list successful-edges)))
  (blank 2)
  (mapc 'treeprint trees)
 )
)

;*******************************************************************************
;                                                                              *
;  FUNCTION: output-rules                                                      *
;  PURPOSE:  To print out a nice listing of the parsings for feature parse.    *
;                                                                              *
;*******************************************************************************

(define (output-rules successful-edges)
;######################################
 (let ((trees (rule-tree-list successful-edges)))
  (blank 2)
  (princ (eval '(pp ,trees)))
 )
)


;*******************************************************************************
;                                                                              *
;  FUNCTION: rule-tree-list                                                    *
;  PURPOSE:  To translate a list of successful edges with their bindings      *
;            into parse trees.                                                 *
;                                                                              *
;*******************************************************************************

(define (rule-tree-list successful-edges)
;########################################
 (cond
   ((eq *sentenceparsed* 'top-down)                    ; top down?
    (mapcar 'make-rule-tree  successful-edges))
   (t (mapcar 'make-rule-tree2 successful-edges))      ; else bottom-up
 )
)


;*******************************************************************************
;                                                                              *
;  FUNCTION: make-rule-tree                                                    *
;  PURPOSE:  To build a complete tree structure from an edge by calling        *
;            make-tree on all the included edges for feature-less grammmar.    *
;                                                                              *
;*******************************************************************************

(define (make-rule-tree edge)
  (cond
   ((is-a-category (category-of edge))                   ; lexical edge?
    (list                                                ; list of
     (rule-of edge)                                      ; rule of edge
     (list (contents-of edge))                           ;  semantics or word
    )
   )
   (t
    (cons
     (rule-of edge)                                      ; rule of edge
     (mapcar 'make-rule-tree (contents-of edge))         ; contained edges
    )
   )
  )
)

;*******************************************************************************
;                                                                              *
;  FUNCTION: make-rule-tree2                                                   *
```

```
;  PURPOSE:  To build a complete tree structure from an edge by calling      *
;            make-tree on all the included edges for a feature grammar.      *
;                                                                            *
;*****************************************************************************

(define (make-rule-tree2 edge)
  (cond
   ((is-a-category (first (category-of edge)))          ; lexical edge?
    (list                                               ; list of
     (rule-of edge)                                     ; rule of edge
     (list (contents-of edge))                          ;  semantics or word
    )
   )
   (t
    (cons
     (rule-of edge)                                     ; rule of edge
     (mapcar 'make-rule-tree2 (mapcar 'car (contents-of edge)))
    )                                                   ;   contained edges
   )
  )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: find-rule-trees                                                 *
;  PURPOSE:  To find all the rules  from the chart by finding all the        *
;            complete rules  starting with edges from start.                 *
;                                                                            *
;*****************************************************************************

(define (find-rule-trees startsymbol chart)
;##########################################
 (let ((start  (first chart))                ; start = first vertex in chart
       (finish (car (last chart))))          ; finish = last vertex in chart
  (find-all-rule-trees
     startsymbol
     finish
     (edges-out start)                      ; examine the edges from start
  )
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: find-all-rule-trees                                             *
;  PURPOSE:  To recurse on the list of all edges from the start vertex and   *
;            test to see if they meet the three conditions required to be    *
;            a legal tree:                                                   *
;              1 -- edge out must have start symbol label.                   *
;              2 -- the right vertex of edge must be the finish vertex.      *
;              3 -- there must be no more required constituents.             *
;  OUTPUT:   Return the list of edges with their rule contents which rep     *
;            the valid parses in the chart.                                  *
;                                                                            *
;*****************************************************************************

(define (find-all-rule-trees     startsymbol finish edges-out-of-start)
;#######################################################################
 (cond
  ((null edges-out-of-start) nil)                   ; finished -> quit
  ((and                                             ; parse good if:
    (eq (first                                      ; 1 first of
         (category-of (head edges-out-of-start)))   ; 1  cat of edge is start
        startsymbol)                                ;    symbol and
    (eq (right-vertex-of (head edges-out-of-start)) ; 2 right vertex is
        finish)                                     ;    last vertex and
    (eq (needed-of (head edges-out-of-start))       ; 3 no needed constituents
        nil))
   (cons
         (list
          (head edges-out-of-start)                 ; keep head of list with
          (contents-of (head edges-out-of-start))   ;  its bindings
         )
         (find-all-rule-trees                       ; examine tail
            startsymbol
            finish
            (tail edges-out-of-start))
   )
  )
  (t (find-all-rule-trees                           ;  else examine tail
        startsymbol
        finish
        (tail edges-out-of-start))
  )
 )
)
```

```
;////////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                               |
;  MODULE:  DEBUG                                                               |
;  PURPOSE: To provide a debugging tool for the programmar as well as a         |
;           method of examing the chart if the user is interested.              |
;                                                                               |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////////

;*******************************************************************************
;                                                                              *
;  FUNCTION: my-debug                                                          *
;  PURPOSE:  To aid myself debugging as well as to aid in process              *
;            explanation by making the chart procedure transparent to the      *
;            caller of the function.                                           *
;                                                                              *
;*******************************************************************************

(define (my-debug)
 (terpri)
 (princ "Please Choose an item below, h for help") (terpri)
 (princ "[VertexN, EdgeN, Chart, Agenda, Grammar, Parses, Store, Quit, Help]?  ")
 (terpri) (princ "What next?  ")
 (setq input (read))
 (let ((command (car (explode input)))
       (number (reverse (cdr (explode input)))))
    (cond
     ((eq command 'v) (cond
                        ((eq *sentenceparsed* 'top-down)
                         (display-vertex-td (item+end 'vertex number)))
                        (t (display-vertex-bu (item+end 'vertex number)))
                        (terpri)))
     ((eq command 'e) (display-edge   (item+end 'edge number)))
     ((eq command 'c) (blank 2)
                      (print-list (append '(CHART CONTAINS:) *chart*))
                      (blank 2))
     ((eq command 'a) (blank 2)
                      (print-list (append '(AGENDA IS:) *agenda*))
                      (blank 2))
     ((eq command 'g) (terpri) (eval `(pp ,*grammar*)))
     ((eq command 'p) (terpri) (princ "Which part of sentence [s np vp etc]?  ")
                      (setq *input* (read))
                      (terpri)
                      (cond
                        ((eq *sentenceparsed* 'top-down)
                          (let ((parse-parent-edges (find-parses *input* *chart*)))
                           (output-parses parse-parent-edges) (terpri)
                           (output-rules parse-parent-edges) (terpri)
                           (output-trees  parse-parent-edges) (terpri)
                           (output-rule-trees parse-parent-edges)))
                         (t
                           (let* ((edges-with-bindings (find-feature-parses *input* *chart*))
                                  (edges (mapcar 'car edges-with-bindings)))
                            (output-parses-features edges-with-bindings) (terpri)
                            (output-rules edges) (terpri)
                            (output-trees-features edges-with-bindings) (terpri)
                            (output-rule-trees edges)))
                         )
     )
     ((eq command 's) (terpri)
      (save (rule-tree-list (mapcar 'car (find-feature-parses 's *chart*)))))
     )
     ((eq command 'h) (terpri)
          (princ "Type first character (LOWER CASE) of item and optional number") (terpri)
          (princ "Examples: Type 'c'  to look at the Chart") (terpri)
          (princ "          Type 'e5' to look at Edge 5 in the Chart") (terpri))
     ((not (eq command 'q)) (terpri)
                            (princ "*** Illegal command -- Type h for help")
                            (terpri))
    )
    (cond ((not (eq command 'q)) (my-debug)))   ; continue if not quit
 )
)

(define (item+end item end)
 (cond
  ((null end) nil)
  ((null (cdr end)) (item+end2 item (car end)))
  (t  (implode (reverse
                        (cons (car end)
                              (reverse (explode (item+end item (cdr end))))))))
  )
 )
)
```

```
(define (item+end2 item end)
    (implode (reverse
        (cons end
              (reverse (explode item))))))

(define (save expr)
 (prog (out)
  (msg N "Outfile:  " )
  (setq out (outfile (read)))
  (pp-form expr out)
  (close out)
  (msg "Item stored" N)
 )
)
```

```
;*****************************************************************************
;                                                                            *
;  FUNCTION: Dictionary Macro definitions (syntax, semantics, etc.)          *
;  PURPOSE:  To perform certain basic operations quickly.                    *
;  INPUT:    Varys, but generally some type of list.                         *
;  OUTPUT:   Some member of the given list or a value such as t or nil.      *
;                                                                            *
;*****************************************************************************

;    dictionary access functions:
;
; a description is of the form:  ( <syntax> <semantics> <realization>)
; where <syntax>      = <category features*>
;       <semantics>   = lambda expression of meaning
;       <realization> = surface expression of lexical entry

(defun syntax       macro (description) '(head ,(args description)))
(defun semantics    macro (description) '(first (tail ,(args description))))
(defun realization macro (description) '(first (tail (tail ,(args description)))))
(defun word-type    macro (description) '(first  (syntax ,(args description))))
(defun word-type-category macro (description) '(second (syntax ,(args description))))
(defun verb-type    macro (description) '(second (syntax ,(args description))))
(defun verb-count   macro (description) '(third (syntax ,(args description))))
(defun verb-tense   macro (description) '(fourth (syntax ,(args description))))
(defun verb-person macro (description) '(fifth  (syntax ,(args description))))
(defun verb-ing     macro (description) '(sixth  (syntax ,(args description))))
(defun noun-type    macro (description) '(second (syntax ,(args description))))
(defun noun-count   macro (description) '(third  (syntax ,(args description))))
(defun noun-gender macro (description) '(fourth (syntax ,(args description))))
(defun adj-type     macro (description) '(second (syntax ,(args description))))

;    Dictionary predicates

(defun word-type-category-p macro (description)
  '(eq ,(args description) (word-type-category ,(args (tail description)))))
(defun verbp macro (description)   (eq 'verb (word-type ,(args description))))
(defun nounp macro (description)  '(eq 'noun (word-type ,(args description))))
(defun adjp  macro (description)  '(eq 'adjective (word-type ,(args description))))
(defun namep macro (description)  '(eq 'proper-noun (word-type ,(args description))))
(defun count? macro (description) '(eq 'count (noun-type ,(args description))))

;    Grammar Rule Macros

(defun rule-name      macro  (ruledescr) '(first  ,(args ruledescr)))
(defun rule-syntax    macro  (ruledescr) '(second ,(args ruledescr)))
(defun rule-semantics macro  (ruledescr) '(third  ,(args ruledescr)))
```

```
;*****************************************************************************
;                                                                            *
;  MODULE:  MAKE DICTIONARY                                                  *
;  PURPOSE: Routines to facilitate the creation and access of dictionary.    *
;                                                                            *
;*****************************************************************************

;*****************************************************************************
;                                                                            *
;  FUNCTION: make-plural                                                     *
;  PURPOSE:  To use explode to make a word plural.                           *
;                                                                            *
;*****************************************************************************

(define (make-plural word)
;#########################
   (implode (reverse (append '(: s) (reverse (explode word)))))
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: my-delete                                                       *
;  PURPOSE:  To delete a pattern from a given list.                          *
;                                                                            *
;*****************************************************************************

(define (my-delete pattern list)
;###############################
 (cond
   ((null list) nil)
   ((equal pattern (car list))              ; first # pattern?
    (my-delete pattern (cdr list))          ;  skip over it, recurse on tail
   )
   (t (cons (car list)                      ; otherwise keep the first and
            (my-delete pattern (cdr list))) ; delete pattern from tail
      )
   )
 )
)


;*****************************************************************************
;                                                                            *
;  FUNCTION: delete-duplicates                                               *
;  PURPOSE:  To delete the duplicates within a list.                         *
;                                                                            *
;*****************************************************************************

(define (delete-duplicates list)
;###############################
 (cond
  ((null list) list)
  (t (cons (car list)
           (delete-duplicates (my-delete (car list) list))
     )
  )
 )
)


;*****************************************************************************
;                                                                            *
;  FUNCTION: a-list                                                          *
;  PURPOSE:  To return the association property list of word.                *
;                                                                            *
;*****************************************************************************

(define (a-list word) (get word 'a-list))
;########################################

;*****************************************************************************
;  NB:                                                                       *
;  I use * surrounding a symbol to distinguish it as a global variable.      *
;                                                                            *
;*****************************************************************************

(define (dictionary property) (get '*dictionary* property))
;##########################################################


;*****************************************************************************
;                                                                            *
;  FUNCTION: dictionary-words                                                *
;  PURPOSE:  To return a list of current dictionary words by recursing       *
;            on the list of lexical categories in the dictionary.            *
;                                                                            *
```

```
;*****************************************************************************

(define (dictionary-words lexical-categories)
;############################################
 (cond
  ((null lexical-categories) nil)
  (t (append (dictionary (car lexical-categories))        ; list of next lex cat
             (dictionary-words (cdr lexical-categories))  ; recurse
     )
  )
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: words-in-dictionary                                             *
;  PURPOSE:  To return a single entry list of dictionary words.              *
;                                                                            *
;*****************************************************************************

(define (words-in-dictionary)
;############################
  (delete-duplicates (dictionary-words (dictionary 'lexical-categories)))
)

;*****************************************************************************
;                                                                            *
;  Set up dictionary as association lists which are stored as property       *
;  lists for each entry.  As each word is added to the dictionary, a-lists   *
;  are built accordingly.                                                    *
;                                                                            *
;*****************************************************************************


;*****************************************************************************
;                                                                            *
;  FUNCTION: make-new-lexical-category                                       *
;  PURPOSE:  checks if the given category already exists in the dictionary.  *
;            if exists already? --> do nothing, return t                     *
;            else add it to the lexicon.                                     *
;                                                                            *
;*****************************************************************************

(define (make-new-lexical-category category)
;###########################################
 (cond
  ((member category (dictionary 'lexical-categories)) t)       ; already?
  (t (putprop '*dictionary*                                    ; else add
              (cons category (dictionary 'lexical-categories)) ; it in
              'lexical-categories
     )
  )
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: make-new-verb-category                                          *
;  PURPOSE:  checks if the given verb category already exists in the         *
;            dictionary.                                                     *
;            if exists already? --> do nothing, return t                     *
;            else - add it to the lexicon                                    *
;                 - check if the category "verb" is in "lexical-categories"  *
;                                                                            *
;*****************************************************************************

(define (make-new-verb-category verb-category)
;##############################################
 (cond
  ((member verb-category (dictionary 'verb-categories)) t)  ; already member?
  (t (putprop '*dictionary*                                 ;  no -> add in
              (cons verb-category (dictionary 'verb-categories))
              'verb-categories)
     (cond                                                  ;  already verb?
      ((member 'verb (dictionary 'lexical-categories)) t)
      (t (putprop                                           ;  no -> add in
            '*dictionary*
            (cons 'verb (dictionary 'lexical-categories))
            'lexical-categories)
      )
     )
  )
 )
)

;*****************************************************************************
```

```
;                                                                              *
;  FUNCTION: add-word-to-lexical-category                                      *
;  PURPOSE:  To add a given word to the lexical category provided.             *
;                                                                              *
;*******************************************************************************

(define (add-word-to-lexical-category word category)
;##################################################
 (cond
  ((member word (dictionary category)) t) ; don't add existing word
  (t (putprop '*dictionary*                ; otherwise add it to
              (cons word (dictionary category)) ; the category
              category
     )
  )
 )
)

;*******************************************************************************
;                                                                              *
;  FUNCTION: update-word-a-list                                                *
;  PURPOSE:  Updates the association property list for the given word by       *
;            adding the provided description to the list of current            *
;            descriptions.                                                     *
;                                                                              *
;*******************************************************************************

(define (update-word-a-list word description)
;#############################################
 (cond ((null (a-list word))                          ; if word is new
        (putprop word (list description) 'a-list)     ;  create a-list for word
       )
       ((member description (a-list word))            ; if word already here
        (tell-user
           '(The word ,word already exists in the dictionary) 1 1)
       )                                              ;  say so
       (t (putprop word                               ; otherwise add new descr
                   (append (list description) (a-list word))
                   'a-list
          )                                           ; of word to the a-list
       )
 )
)

;*******************************************************************************
;                                                                              *
;  FUNCTION:  make-dictionary-entry                                            *
;  PURPOSE:   To make a dictionary entry from the given list.  List is in      *
;             the following form:                                              *
;                                                                              *
;             ( <word-name> (( <syntax> )  ( <semantics> ) <realization>))     *
;                                                                              *
;*******************************************************************************

(define (make-dictionary-entry list)
;###################################
 (let* ((word (head list))                        ; the word is the first item
        (description (tail list))                 ; the end is the tail
        (syntax (first description))              ; the syntax is first of end
        (category (first syntax)))                ; the category is first of syntax
  (cond
   ((eq category 'verb)                           ; is it a verb?
    (make-new-verb-category (second syntax))      ;  keep its verb type
   )
   (t (make-new-lexical-category category))       ; add lex cat if non-existant
  )
  (add-word-to-lexical-category word category)    ; add word if new
  (update-word-a-list word description)           ; update the a-list of word
 )
)


;###############################################################################
;###############################################################################
;                                                                              #
;  dictionary selectors, mutators, and displayers.                             #
;                                                                              #
;###############################################################################
;###############################################################################

;*******************************************************************************
;                                                                              *
;  FUNCTION: pretty-print-a-list                                               *
;  PURPOSE:  To print a gorgeous looking association list for word.            *
;                                                                              *
;*******************************************************************************
```

```
(define (pretty-print-a-list word)
;#################################
    (eval '(pp ,(cons word (a-list word))))   ; pp word with assoc list
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: entry-exists                                                    *
;  PURPOSE:  To return t if an item has an a-list, nil otherwise.            *
;                                                                            *
;*****************************************************************************

(define (entry-exists item)
;##########################
   (cond ((null (a-list item)) nil)           ; return nil if no entry
         (t t)                                ;  t otherwise
   )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: find-entries                                                    *
;  PURPOSE:  To show the word indicated.                                     *
;                                                                            *
;*****************************************************************************

(define (find-entries item) (show-word item))
;############################################

;*****************************************************************************
;                                                                            *
;  FUNCTION: show-word                                                       *
;  PURPOSE:  To print out nicely the word or to say that it does not exists  *
;  INPUT:    a word                                                          *
;  OUTPUT:   pretty-print-a-list if word exists or error message             *
;                                                                            *
;*****************************************************************************

(define (show-word word)
;#######################
 (cond ((entry-exists word)            ; if there is an entry return
        (pretty-print-a-list word))    ;  then show user a friendly list
       (t                              ; otherwise say you can't find it
         (tell-user (append1 '(There is no dictionary entry for) word) 1 1)
       )
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: lex-cat-in-a-list                                               *
;  PURPOSE:  To test if a given a-list for some word contains a description  *
;            of given lexical category (ie return T if a word is of given    *
;            lexical type).                                                  *
;  INPUT:    a lexical category and the a-list for some word                 *
;  OUTPUT:   t or nil                                                        *
;                                                                            *
;*****************************************************************************

(define (lex-cat-in-a-list? category a-list)
;###########################################
 (cond ((null (car a-list)) nil)                    ; empty head a-list --> no cat in a-list
       ((eq (caaar a-list) category) t)             ; next item in category? --> t
       (t (lex-cat-in-a-list? category (cdr a-list))) ; otherwise examine tail
 )
)
;*****************************************************************************
;                                                                            *
;  FUNCTION: is-a                                                            *
;  PURPOSE:  To test if a given word is an example of type category.         *
;  INPUT:    a lexical category and a word                                   *
;  OUTPUT:   t or nil                                                        *
;                                                                            *
;*****************************************************************************

(define (is-a category word)
;############################
 (cond
   ((is-a-category category)                        ; category known?
    (lex-cat-in-a-list? category (a-list word))     ;  check a-list of word
   )                                                ; otherwise not known
   ((verb? category)                                ; verb?
    (is-of-type category (a-list word))             ;  check a-list
   )
   (t (tell-user '(,category is not a know lexical category 1 1)))
```

```
 )
)

;*************************************************************************
;                                                                        *
;  FUNCTION: is-a-noun                                                   *
;  PURPOSE:  To determine if a given symbol is a noun in the dictionary  *
;  INPUT:    a word (atom)                                               *
;  OUTPUT:   t or nil                                                    *
;                                                                        *
;*************************************************************************

(define (is-a-noun word)
;########################
   (lex-cat-in-a-list? 'noun (a-list word))      ; is the lexical category
)                                                ;  noun in the a-list?

;*************************************************************************
;                                                                        *
;  FUNCTION: is-a-verb                                                   *
;  PURPOSE:  To determine if a given symbol is a verb in the dictionary  *
;  INPUT:    a word (atom)                                               *
;  OUTPUT:   t or nil                                                    *
;                                                                        *
;*************************************************************************

(define (is-a-verb word)
;########################
   (lex-cat-in-a-list? 'verb (a-list word))      ; is the lexical category
)                                                ;  verb in the a-list?

;*************************************************************************
;                                                                        *
;  FUNCTION: list-dictionary-words                                       *
;  PURPOSE:  To list the dictionary descriptions of a given list of words.*
;  INPUT:    list of words                                               *
;  OUTPUT:   Dictionary description of each word in word list or appropriate*
;            message if a word in the list in not in the dictionary.     *
;                                                                        *
;*************************************************************************

(define (list-dictionary-words wordlist)
;#######################################
 (cond ((null wordlist) nil)                  ; stop if no more words
       ((null (a-list (car wordlist)))        ; if no next word say so
        (tell-user
           (append1 '(There is no dictionary entry for) (car wordlist)) 1 1)
       )
       (t                                     ; list next word with
          (eval '(pp ,(cons (car wordlist) (a-list (car wordlist)))))
          (list-dictionary-words (cdr wordlist))     ;  recurse on tail
       )
 )
)

;*************************************************************************
;                                                                        *
;  FUNCTION: list-dictionary-words-short                                 *
;  PURPOSE:  To list the words in a given list of words.                 *
;  INPUT:    list of words                                               *
;  OUTPUT:   the list of words                                           *
;                                                                        *
;*************************************************************************

(define (list-dictionary-words-short wordlist)
;##############################################
 (cond ((null wordlist) nil)                 ; no more words --> stop
       (t (msg (car wordlist) N)             ; otherwise show first
          (list-dictionary-words-short (cdr wordlist)) ; recurse on tail
       )
 )
)

;*************************************************************************
;                                                                        *
;  FUNCTION: filter-a-list                                               *
;  PURPOSE:  To return the list of dictionary descriptions which appear in *
;            the given a-list of some word and are of given lexical category*
;  INPUT:    a lexical category and association list of a word           *
;  OUTPUT:   Dictionary description of each entry in a-list which belongs *
;            to the given lexical category.                              *
;                                                                        *
;*************************************************************************

(define (filter-a-list category a-list)
;######################################
```

```
 (cond ((null a-list) nil)                          ; no more words --> stop
       ((eq (word-type (head a-list)) category)     ; if next descr in category -> keep
        (append
          (list (head a-list))
          (filter-a-list category (tail a-list)))
       )
       (t (filter-a-list category (tail a-list))) ; otherwise check tail (recurse)
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: list-category-descriptions-only                                 *
;  PURPOSE:  To print all the dictionary descriptions of lexical type        *
;            category which appear in the a-list for each word in wordlist   *
;  INPUT:    a lexical category and a list of words                          *
;  OUTPUT:   All the dictionary descriptions for each word in wordlist       *
;            which belongs to the given lexical category.                    *
;                                                                            *
;*****************************************************************************

(define (list-category-descriptions-only category wordlist)
;##########################################################
 (cond ((null wordlist) nil)                ; no more words --> stop
       (t (eval '(pp ,(cons (car wordlist)
                 (filter-a-list category (a-list (car wordlist))) ) )); first
          (list-category-descriptions-only category (cdr wordlist)) ; recurse on tail
       )
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: list-lexical-categories                                         *
;  PURPOSE:  To print a list of the lexical categories along with the words  *
;            which belong to each category and their corresponding           *
;            dictionary descriptions.                                        *
;  INPUT:    a list of lexical categories                                    *
;  OUTPUT:   Printout of dictionary descriptions for each word in            *
;            dictionary, listed by lexical category.                         *
;                                                                            *
;*****************************************************************************

(define (list-lexical-categories categorylist)
;#############################################
 (cond ((null categorylist) nil)            ; no more words --> stop
       (t                                   ; otherwise
          (tell-user (list (make-plural (car categorylist))) 1 1)  ; show category
          (list-category-descriptions-only  (car categorylist)
              (dictionary (car categorylist))
          )
          (list-lexical-categories (cdr categorylist)) ; recurse on tail
       )
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: list-lexical-categories-short                                   *
;  PURPOSE:  To print a list of the lexical categories along with the words  *
;            which belong to each category                                   *
;  INPUT:    a list of lexical categories                                    *
;  OUTPUT:   Printout of each word in dictionary, listed by lex category.    *                    *
;                                                                            *
;*****************************************************************************

(define (list-lexical-categories-short categorylist)
;###################################################
 (cond ((null categorylist) nil)            ; no more words --> stop
       (t
          (tell-user (list (make-plural (car categorylist))) 1 1)   ; print lex cat
          (list-dictionary-words-short (dictionary (car categorylist)))          ; otherwise show f:
st
          (list-lexical-categories-short (cdr categorylist)) ; recurse on tail
       )
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: show-dictionary-formats                                         *
;  PURPOSE:  To display the various formats available for show-dictionary.   *
;                                                                            *
;*****************************************************************************

(define (show-dictionary-formats)
```

```
;#################################
  (tell-user '(Options for Print Dictionary Command:) 2 1)
  (tell-user '(s - Simple or Short list of alphabetized words) 1 0 5)
  (tell-user '(a - All information for words in aphabetical order) 1 0 5)
  (tell-user '(l - Lexically ordered list of words) 1 0 5)
  (tell-user '(    (ie nouns - verbs - determiners etc)) 1 0 9)
  (tell-user '(c - Complete word descriptions by lexical category) 1 0 5)
  (tell-user '(? - Describe the show dictionary formats) 1 0 5)
)

;*****************************************************************************
;                                                                             *
;  FUNCTION: show-dictionary                                                  *
;  PURPOSE:  To print out a pretty list of dictionary entries, possibly       *
;            with their descriptions, in the order the user desires most.     *
;  INPUT:    an optional format argument                                      *
;  OUTPUT:   Printout of dictionary words.  The exact format of output is     *
;            determined by the option chosen.  If no option is provided,      *
;            the output defaults to a simple list of dictionary words.        *
;  OPTIONS:                                                                   *
;                s -- Simple or Short list of alphabetized words              *
;                a -- All information for words in aphabetical order          *
;                l -- Lexically ordered list of words                         *
;                     (ie nouns, verbs, determiners, etc)                     *
;                c -- Complete word descriptions by lexical category          *
;                ? -- Describe the show dictionary formats                    *
;                                                                             *
;           nil option will default to 's and incorrect option will hiccup    *
;                                                                             *
;*****************************************************************************

(define (show-dictionary &optional format)
;########################################
 (cond
  ((eq format '?)
   (show-dictionary-formats)                    ; describe formats
  )
  ((not (member format '(s a l c ? nil)))                ; invalid format?
   (tell-user '(*** ,format is not a valid option) 1 1)  ;   --> hiccup
   (show-dictionary-formats)                            ; describe formats
  )
  ((null (words-in-dictionary))                         ; no dictionary?
   (blank 2)
   (princ "*** There are no current dictionary entries.") ;  --> complain
   (blank 2)
  )
  (t                                            ; else show it by format
   (tell-user '(The current dictionary words include: ) 1 2 5)
   (cond
    ((or (eq format 's) (equal format nil))     ; user wants Short format?
     (list-dictionary-words-short
       (words-in-dictionary))
    )
    ((eq format 'a)                              ; user wants All format?
     (list-dictionary-words
       (words-in-dictionary))
    )
    ((eq format 'l)                              ; user wants lexicon long?
     (list-lexical-categories
       (dictionary 'lexical-categories))
    )
    ((eq format 'c)                               ; short lexicon listing?
       (list-lexical-categories-short
         (dictionary 'lexical-categories))
    )
   )                                              ; close format cond
  )                                               ; close otherwise
 )                                                ; close cond
)                                                 ; end show-dictionary

;*****************************************************************************
;                                                                             *
;  FUNCTION: is-a-category                                                    *
;  PURPOSE:  to tell if a given symbol is a lexical category                  *
;  INPUT:    symbol (potentially a lexical category)                          *
;  OUTPUT:   t or nil                                                         *
;                                                                             *
;*****************************************************************************

(define (is-a-category item)
;###########################
 (cond
  ((member item (dictionary 'lexical-categories))) ; regular lexical category
  ((verb? item))                                   ; else a type of verb?
 )
)
```

```
(define (verb? category)
;#######################
 (cond ((member category (dictionary 'verb-categories))))
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: my-random                                                       *
;  PURPOSE:  to use the random number generator (which is not so random)     *
;            to produce a (relatively) more random number in given range     *
;  INPUT:    range maximum                                                   *
;  OUTPUT:   random number between 0 (not inclusive) and given maximum       *
;                                                                            *
;*****************************************************************************

(define (my-random n)
;####################
 (add1 (quotient (random (times 100 n)) 100))
)


;*****************************************************************************
;                                                                            *
;  FUNCTION: find-example-of                                                 *
;  PURPOSE:  To randomly choose a word within some lexical category          *
;  INPUT:    lexical category                                                *
;  OUTPUT:   Error message if category non-existant or randomly selected     *
;            word from within given lexical category.                        *
;                                                                            *
;*****************************************************************************

(define (find-example-of category)
;#################################
 (cond
   ((not (is-a-category category)) ; not a category?
    (tell-user                                  ;  --> complain
       '(There are no words of the lexical category ,category
                in the dictionary) 1 1)
   )
   ((verb? category)
    (pick-random-element-from (filter-type (dictionary 'verb) category))
   )                                ; restrict the type if need be
   (t (pick-random-element-from (dictionary category))) ; pick random category word
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: is-of-type                                                      *
;  PURPOSE:  Tests the a-list of a word to determine if it is of the given   *
;            type within its lexical category.                               *
;  INPUT:    a-list of a word                                                *
;  OUTPUT:   t if of given type, nil otherwise                               *
;                                                                            *
;*****************************************************************************

(define (is-of-type a-list type)
;###############################
 (cond
  ((null a-list) nil)                           ; empty? --> not of type
  ((word-type-category-p type (head a-list)) t) ; next word ok type? --> t
  (t (is-of-type (tail a-list) type))           ; otherwise check rest
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: filter-type                                                     *
;  PURPOSE:  To select only those words of a particular type within a        *
;            list of words from a lexical category.                          *
;  INPUT:    list of words in lexical category                               *
;  OUTPUT:   only those words in lexical category list, also of given type.  *
;                                                                            *
;*****************************************************************************

(define (filter-type words-in-category type)
;###########################################
 (cond
  ((null words-in-category) nil)
  ((is-of-type (a-list (head words-in-category)) type) ; next word ok type?
   (cons                                               ; make new list
     (head words-in-category)                          ;  keep first word
     (filter-type (tail words-in-category) type)       ;  filter rest
   )
  )
```

```
  (t (filter-type (tail words-in-category) type))      ; otherwise filter rest
 )
)

;*****************************************************************************
;                                                                             *
;  FUNCTION: pick-random-element-from                                         *
;  PURPOSE:  To randomly choose an element from some list                     *
;  INPUT:    a list                                                           *
;  OUTPUT:   A random element of the list.  The function nthelem is used      *
;            which returns the nth element in lyst starting with 1.          *
;            The function nth does the same starting with 0.                  *
;                                                                             *
;*****************************************************************************

(define (pick-random-element-from lyst)
;######################################
  (nthelem (my-random (length lyst)) lyst)
)

;*****************************************************************************
;                                                                             *
;  FUNCTION: erase-words                                                      *
;  PURPOSE:  To empty out all the current a-list properties for words         *
;  INPUT:    list of words                                                    *
;  OUTPUT:   nil                                                              *
;                                                                             *
;*****************************************************************************

(define (erase-words wordlist)
;##############################
 (cond
  ((null wordlist) nil)                          ; no more? --> finished
  (t                                             ; otherwise
   (remprop (head wordlist) 'a-list)             ; remove a-list
   (erase-words (tail wordlist))                 ; recurse tail
  )
 )
)

;*****************************************************************************
;                                                                             *
;  FUNCTION: erase-lexical-categories                                         *
;  PURPOSE:  To empty out all the current lexical category property lists     *
;  INPUT:    list of lexical categories                                       *
;  OUTPUT:   nil                                                              *
;                                                                             *
;*****************************************************************************

(define (erase-lexical-categories lexical-categories)
;####################################################
 (cond
  ((null lexical-categories) nil)                    ; no more? --> finished
  (t                                                 ; otherwise
   (erase-words (dictionary (head lexical-categories))); erase words in lex cat
   (erase-lexical-categories (tail lexical-categories)); recurse tail
  )
 )
)

;*****************************************************************************
;                                                                             *
;  FUNCTION: erase-dictionary                                                 *
;  PURPOSE:  To erase the dictionary by emptying out each lexical category    *
;            property of *dictionary* (ie noun, verb, determiner, etc), as    *
;            well as the property "lexical-category" of *dictionary* itself.  *
;  INPUT:    nil                                                              *
;  OUTPUT:   nil                                                              *
;                                                                             *
;*****************************************************************************

(define (erase-dictionary)
;##########################
  (erase-lexical-categories (dictionary 'lexical-categories))
  (setplist '*dictionary* nil)                       ; erase plist
)                                                    ; on dictionary
```

```
;////////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                               |
;  MODULE:   DICTIONARY LOOKUP                                                  |
;  PURPOSE:  To provide proper entries for provided word in format:             |
;            ( word-root ( <syntax> <semantics> <realization> ))                |
;  OWNER:    copywrite Mark T. Maybury, June, 1987.                             |
;                                                                               |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////////

;-------------------------------------------------------------------------------
;  FUNCTION: look-up                                                            |
;  PURPOSE:  To search the dictionary for an occurance of the given word,       |
;            returning appropriate entries.                                     |
;  INPUT:    A word in "raw" format (ie with any legal suffix or                |
;            conjugation.)                                                      |
;  METHOD:   The property lists are filtered to assure compatability with       |
;            provided features. (ie if the word is plural then change           |
;            the standard entry accordingly)                                    |
;                                                                               |
;-------------------------------------------------------------------------------

(define (look-up word)
;---------------------
 (let ((entries (a-list word)))
  (cond
   ((null entries) nil)
   (t (cons word entries))
  )
 )
)

;-------------------------------------------------------------------------------
;  FUNCTION: look-up-verb                                                       |
;  PURPOSE:  To search the dictionary for an occurance of the given verb,       |
;            returning appropriate entries with the word verb deleted for       |
;            compatability with grammar.                                        |
;-------------------------------------------------------------------------------

(define (look-up-verb word)
;--------------------------
 (let ((entries (delete-word-verb (a-list word))))
  (cond
   ((null entries) nil)
   (t (cons word entries))
  )
 )
)

;-------------------------------------------------------------------------------
; FUNCTION: delete-word-verb                                                    |
; PURPOSE:  to remove the word verb from the lexical entry list to allow       |
;           for consistency with grammar features.                              |
;-------------------------------------------------------------------------------

(define (delete-word-verb a-list)
;--------------------------------
 (cond
  ((null a-list) nil)
  (t
   (cons
    (list (my-delete 'verb (syntax (head a-list)))
          (semantics (head a-list))
          (realization (head a-list)))
    (delete-word-verb (tail a-list))
   )
  )
 )
)

;*******************************************************************************
;                                                                              *
;  FUNCTION: make-verbs-past-participle   ("eat" -> "eaten")                   *
;  PURPOSE:  To return the list of dictionary descriptions which appear in     *
;            the given a-list of some word, filtered so that the verbs are     *
;            past-participle.                                                  *
;  INPUT:    the association list for the word                                 *
;  OUTPUT:   The modified a-list                                               *
;                                                                              *
;*******************************************************************************

(define (make-verbs-past-participle a-list)
;##########################################
 (cond
  ((null a-list) nil)
```

```
((eq (word-type (head a-list)) 'verb)                    ; change only if verb
 (append                                                 ;  attach fixed head to tail
  (list                                                  ;  fix head
   (append
      (append1
         (my-delete 'verb
          (subst 'past 'pres (syntax (head a-list))))    ; change to past-participle
         'en)
      (list (semantics (head a-list)))                   ; save semantics
      (list (realization (head a-list)))                 ; save realisation
   )
  )
  (make-verbs-past-participle (tail a-list))             ;  check tail
 )
)
(t (make-verbs-past-participle (tail a-list)))           ; else check only tail
)                                                        ; (throws out non verbs)
)
```

```
;*****************************************************************************
;                                                                            *
;  MODULE:  LOOKUP                                                           *
;  PURPOSE: To provide support functions for dictionary.                     *
;                                                                            *
;*****************************************************************************

;*****************************************************************************
;                                                                            *
;  FUNCTION: singular-noun?                                                  *
;  PURPOSE:  To determine if a given description for some word is both       *
;            in the lexical category noun and of singular persuasion.        *
;  INPUT:    description (an element of some word's a-list)                  *
;  OUTPUT:   t or nil                                                        *
;                                                                            *
;*****************************************************************************

(define (singular-noun? description)
;###################################
  (and (eq (word-type description)  'noun)       ; lexical category noun?
       (eq (noun-count description) 'sing3p)     ; singular?
  )
)

(define (mass-noun? description)
;###############################
  (and (eq (word-type description) 'noun)        ; lexical category noun?
       (eq (noun-type description) 'mass)        ; mass noun?
  )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: 3ps-verb?                                                       *
;  PURPOSE:  To determine if the description is of a verb in the third       *
;            person singular conjugation.                                    *
;  INPUT:    description (an element of some word's a-list)                  *
;  OUTPUT:   t or nil                                                        *
;                                                                            *
;*****************************************************************************

(define (3ps-verb? description)
;##############################
  (and (eq (word-type description)  'verb)        ; lexical category verb?
       (or (eq (verb-count description) 'sing3p)   ; singular3p?
           (eq (verb-count description) 'sing))    ; singular?
;       (eq (verb-person description)  'p3)        ; third person?
  )
)

;#############################################################################
;#############################################################################
;########                  A-LIST FILTERS                     ################
;#############################################################################
;#############################################################################

;*****************************************************************************
;                                                                            *
;  FUNCTION: make-nouns-plural-and-verbs-3ps                                 *
;  PURPOSE:  To return the list of dictionary descriptions which appear in   *
;            the given a-list of some word, filtered so that the nouns are   *
;            plural and the verbs are third person singular.                 *
;  INPUT:    the association list for the word                               *
;  OUTPUT:   The modified a-list                                             *
;                                                                            *
;*****************************************************************************

(define (make-nouns-plural-and-verbs-3ps a-list)
;###############################################
 (cond
  ((null a-list) nil)                                  ; no more words --> stop
  ((singular-noun? (head a-list))                      ; next descr sing noun?
   (append                                             ;  attach fixed head to tail
     (list                                             ;  fix head
      (cons (subst 'plur 'sing3p (syntax (head a-list))); change to plural
            (list (semantics (head a-list)))           ; save rest
      )
     )
     (make-nouns-plural-and-verbs-3ps (tail a-list))   ;  check tail
   )
  )
  ((mass-noun? (head a-list))                          ; mass noun?
   (make-nouns-plural-and-verbs-3ps (tail a-list))     ;  drop head, recurse tail
  )
;  ((eq (verb-person (head a-list)) 'p1)                ; first person?
;   (make-nouns-plural-and-verbs-3ps (tail a-list))     ;  drop head, recurse tail
```

```
;  )
  ((not (3ps-verb? (head a-list)))                          ; next descr not3ps verb?
   (append                                                  ;  attach fixed head to tail
    (list
      (cons
        (my-delete 'verb
        (subst 'sing3p 'plur                                ; make singular
;         (subst 'p3 'p1                                    ;  make sure person
;          (subst 'p3 'p2                                   ;  is 3rd on
           (syntax (head a-list)))))) ;))                   ;   on head
        (list (semantics (head a-list)))                    ; save rest
      )
    )
    (make-nouns-plural-and-verbs-3ps (tail a-list))         ;  check tail
   )
  )
  (t                                                        ; otherwise
   (append
    (delete-word-verb (list (head a-list)))                 ;  keep head as it is and
    (make-nouns-plural-and-verbs-3ps (tail a-list))         ;  recurse on tail
   )
  )
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: make-verbs-past-tense                                           *
;  PURPOSE:  To return the list of dictionary descriptions which appear in   *
;            the given a-list of some word, filtered so that the verbs are   *
;            past tense.                                                     *
;  INPUT:    the association list for the word                               *
;  OUTPUT:   The modified a-list                                             *
;                                                                            *
;*****************************************************************************

(define (make-verbs-past-tense a-list)
;####################################
 (cond
  ((null a-list) nil)
  ((eq (word-type (head a-list)) 'verb)                     ; change only if verb
   (append                                                  ;  attach fixed head to tail
    (list                                                   ;  fix head
     (cons
          (my-delete 'verb
           (subst 'past 'pres (syntax (head a-list))))     ; change to past
          (list (semantics (head a-list)))                  ; save rest
     )
    )
    (make-verbs-past-tense (tail a-list))                   ;  check tail
   )
  )
  (t (make-verbs-past-tense (tail a-list)))                 ; else check only tail
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: make-verbs-n*t                                                  *
;  PURPOSE:  To return the list of dictionary descriptions which appear in   *
;            the given a-list of some word, filtered so that the verbs are   *
;            n*t which stands for n't or negative.  (ex don't)               *
;  INPUT:    the association list for the word                               *
;  OUTPUT:   The modified a-list                                             *
;                                                                            *
;*****************************************************************************

(define (make-verbs-n*t a-list)
;##############################
 (cond
  ((null a-list) nil)
  ((eq (word-type (head a-list)) 'verb)                     ; change only if verb
   (append                                                  ;  attach fixed head to tail
    (list                                                   ;  fix head
     (cons
          (append1 (my-delete 'verb (syntax (head a-list)))
                   'n*t)                                    ; syntax + n't
          (list (semantics (head a-list)))                  ; save rest
     )
    )
    (make-verbs-n*t (tail a-list))                          ;  check tail
   )
  )
  (t (make-verbs-n*t (tail a-list)))                        ; else check only tail
 )                                                          ; (throws out non verbs)
)
```

```
;*****************************************************************************
;                                                                            *
;  FUNCTION: make-verbs-past-participle   ("eat" -> "eaten")                 *
;  PURPOSE:  To return the list of dictionary descriptions which appear in   *
;            the given a-list of some word, filtered so that the verbs are   *
;            past-participle.                                                *
;  INPUT:    the association list for the word                               *
;  OUTPUT:   The modified a-list                                             *
;                                                                            *
;*****************************************************************************

(define (make-verbs-past-participle a-list)
;##########################################
 (cond
  ((null a-list) nil)
  ((eq (word-type (head a-list)) 'verb)                ; change only if verb
   (append                                             ;  attach fixed head to tail
    (list                                              ;  fix head
     (cons
        (append1
           (my-delete 'verb
            (subst 'past 'pres (syntax (head a-list))))); change to past-participle
           'en)
        (list (semantics (head a-list)))               ; save semantics
     )
    )
    (make-verbs-past-participle (tail a-list))         ;  check tail
   )
  )
  (t (make-verbs-past-participle (tail a-list)))       ; else check only tail
 )                                                     ; (throws out non verbs)
)


;*****************************************************************************
;                                                                            *
;  FUNCTION: make-verbs-ing          ("snore" -> "snoring")                  *
;  PURPOSE:  To return the list of dictionary descriptions which appear in   *
;            the given a-list of some word, filtered so that the verbs are   *
;            ing.                                                            *
;  INPUT:    the association list for the word                               *
;  OUTPUT:   The modified a-list                                             *
;                                                                            *
;*****************************************************************************

(define (make-verbs-ing a-list)
;##########################################
 (cond
  ((null a-list) nil)
  ((eq (word-type (head a-list)) 'verb)                ; change only if verb
   (append                                             ;  attach fixed head to tail
    (list                                              ;  fix head
     (cons
        (append1                                       ; add past-participle
          (my-delete 'verb (syntax (head a-list)))
          'ing)
        (list (semantics (head a-list)))               ; save semantics
     )
    )
    (make-verbs-ing (tail a-list))                     ;  check tail
   )
  )
  (t (make-verbs-ing (tail a-list)))                   ; else check only tail
 )                                                     ; (throws out non verbs)
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: make-nouns-and-names-singular-possessive                        *
;            ("boy"  -> "boy's")  ("mark" -> "mark's")                       *
;  PURPOSE:  To return the list of dictionary descriptions which appear in   *
;            the given a-list of some word, filtered so that the nouns       *
;            or names are singular possessive.                               *
;  INPUT:    the association list for the word                               *
;  OUTPUT:   The modified a-list                                             *
;                                                                            *
;*****************************************************************************

(define (make-nouns-and-names-singular-possessive a-list)
;########################################################
 (cond
  ((null a-list) nil)
  ((or (eq (word-type (head a-list)) 'noun)            ; change only if noun
       (eq (word-type (head a-list)) 'proper-noun))    ;  or name
   (append                                             ;  attach fixed head to tail
```

```
        (list                                                   ;  fix head
         (cons                                                  ;  make new list
               (append1
                 (syntax (head a-list))                         ;   keep old syntax
                 'poss)                                         ;   add possessive
               (list (semantics (head a-list)))                 ;   save rest
         )
        )                                                       ;  check tail
        (make-nouns-and-names-singular-possessive (tail a-list))
       )
      )                                                         ; else check only tail
      (t (make-nouns-and-names-singular-possessive (tail a-list)))
     )                                                          ; (throws out non verbs)
    )

    ;*****************************************************************************
    ;                                                                             *
    ;  FUNCTION: make-nouns-plural-possessive  ("boy" -> "boys'")                 *
    ;  PURPOSE:  To return the list of dictionary descriptions which appear in    *
    ;            the given a-list of some word, filtered so that the nouns are    *
    ;            plural possessive.                                               *
    ;  INPUT:    the association list for the word                                *
    ;  OUTPUT:   The modified a-list                                              *
    ;                                                                             *
    ;*****************************************************************************

    (define (make-nouns-plural-possessive a-list)
    ;############################################
     (cond
      ((null a-list) nil)
      ((eq (word-type (head a-list)) 'noun)                     ; change only if noun
       (append                                                  ;  attach fixed head to tail
        (list                                                   ;  fix head
         (cons                                                  ;  make new list
               (append1
                 (subst 'plur 'sing3p (syntax (head a-list)));  make syntax plural
                 'poss)                                         ;   add: sing-poss
               (list (semantics (head a-list)))                 ;   save rest
         )
        )
        (make-nouns-plural-possessive (tail a-list))            ;  check tail
       )
      )
      (t (make-nouns-plural-possessive (tail a-list)))          ; else check only tail
     )                                                          ; (throws out non verbs)
    )

    ;*****************************************************************************
    ;                                                                             *
    ;  FUNCTION: make-adjectives-and-nouns-adverbs  ("slow" -> "slowly")          *
    ;  PURPOSE:  To return the list of dictionary descriptions which appear in    *
    ;            the given a-list of some word, filtered so that the adjectives   *
    ;            and nouns become adverbs.                                        *
    ;  INPUT:    the association list for the word                                *
    ;  OUTPUT:   The modified a-list                                              *
    ;                                                                             *
    ;*****************************************************************************

    (define (make-adjectives-and-nouns-adverbs a-list)
    ;#################################################
     (cond
      ((null a-list) nil)
      ((or
        (eq (word-type (head a-list)) 'adjective)               ; change only if adjective
        (eq (word-type (head a-list)) 'noun))                   ; or noun
       (append                                                  ;  attach fixed head to tail
        (list                                                   ;  fix head
         (cons '(adverb)                                        ; change syntax to adverb
               (list                                            ; semantics is of form
                (list                                           ;  (L (_e) (adverb _e))
                 'L
                 '(_e)
                  (list (semantics (head a-list)) '_e)
                )
               )
         )                                                      ; convert semantics to
        )                                                       ;  adverb type
        (make-adjectives-and-nouns-adverbs (tail a-list))  ;  check tail
       )
      )
      (t (make-adjectives-and-nouns-adverbs (tail a-list))) ; else check only tail
     )
    )

    ;*****************************************************************************
    ;                                                                             *
```

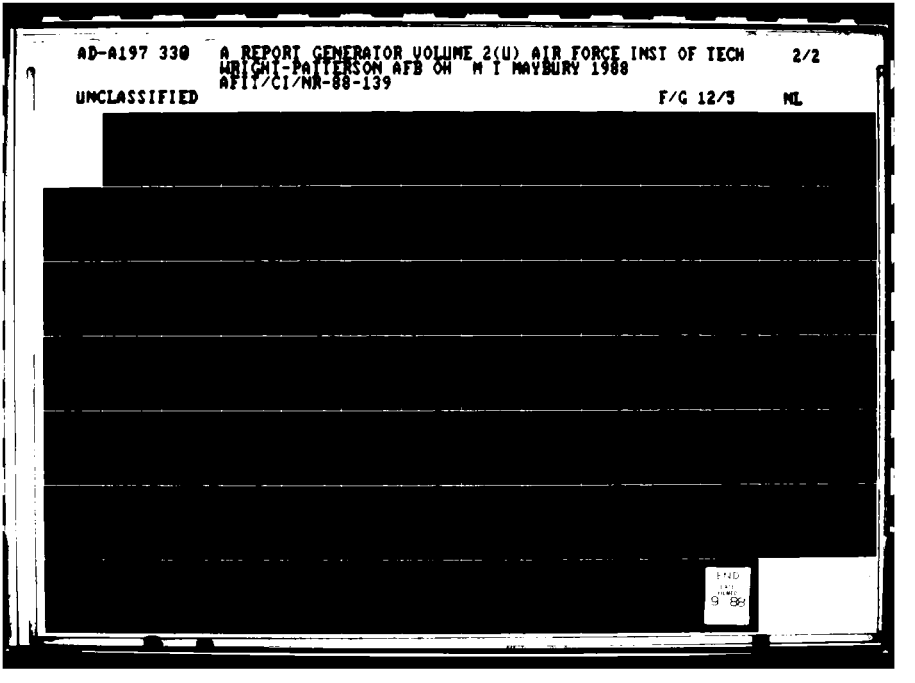
```
;  FUNCTION: make-adjectives-comparative (for words like "slow" -> "slower")*
;  PURPOSE:  To return the list of dictionary descriptions which appear in  *
;            the given a-list of some word, filtered so that the adjectives *
;            become comparative.                                            *
;  INPUT:    the association list for the word                              *
;  OUTPUT:   The modified a-list                                            *
;                                                                           *
;***************************************************************************

(define (make-adjectives-comparative a-list)
;###########################################
 (cond
  ((null a-list) nil)
  ((eq (word-type (head a-list)) 'adjective)    ; change only if adjective
   (append                                      ;  attach fixed head to tail
    (list                                       ;  fix head
      (cons
        (subst 'comparative 'attributive        ; attrib to comparative
               (syntax (head a-list)))          ;  on syntax list
        (list (semantics (head a-list))))       ; save rest
    )
    (make-adjectives-comparative (tail a-list))     ;  check tail
   )
  )
  (t (make-adjectives-comparative (tail a-list)))   ; else check only tail
 )
)

;***************************************************************************
;                                                                           *
;  FUNCTION: make-adjectives-superlative (for words like "big" -> "biggest")*
;  PURPOSE:  To return the list of dictionary descriptions which appear in  *
;            the given a-list of some word, filtered so that the adjectives *
;            become superlative.                                            *
;  INPUT:    the association list for the word                              *
;  OUTPUT:   The modified a-list                                            *
;                                                                           *
;***************************************************************************

(define (make-adjectives-superlative a-list)
;###########################################
 (cond
  ((null a-list) nil)
  ((eq (word-type (head a-list)) 'adjective)    ; change only if adjective
   (append                                      ;  attach fixed head to tail
    (list                                       ;  fix head
      (cons
        (subst 'superlative 'attributive        ; attrib to superlative
               (syntax (head a-list)))          ;  on syntax list
        (list (semantics (head a-list))))       ; save rest
    )
    (make-adjectives-superlative (tail a-list))     ;  check tail
   )
  )
  (t (make-adjectives-superlative (tail a-list)))   ; else check only tail
 )
)

;***************************************************************************
;                                                                           *
;  FUNCTION: make-nouns-adjectives (for words like "fool" -> "foolish")     *
;  PURPOSE:  To return the list of dictionary descriptions which appear in  *
;            the given a-list of some word, filtered so that the nouns      *
;            become attributive adjectives.                                 *
;  INPUT:    the association list for the word                              *
;  OUTPUT:   The modified a-list                                            *
;                                                                           *
;***************************************************************************

(define (make-nouns-adjectives a-list)
;#####################################
 (cond
  ((null a-list) nil)
  ((eq (word-type (head a-list)) 'noun)         ; change only if noun
   (append                                      ;  attach fixed head to tail
    (list                                       ;  fix head
      (cons
        '(adjective attributive)                ; new syntax
         (list (semantics (head a-list))))      ; semantics of N and ADJ same
    )
    (make-nouns-adjectives (tail a-list))       ;  check tail
   )
  )
  (t (make-nouns-adjectives (tail a-list)))     ; else check only tail
 )
)
```

```
;*****************************************************************************
;                                                                             *
;  FUNCTION: word-has-ending?                                                 *
;  PURPOSE:  To determine if the provided word ends with the given ending.    *
;            This is done by comparing the last letter of the ending with     *
;            the last letter of the word until you find that a letter does    *
;            not match (return nil) or until you run out of characters        *
;            (return t since routine has recognized endings as equivalent)    *
;            This routine is tail recursive.                                  *
;  INPUT:    a word (atom) and an ending (atom)                               *
;  OUTPUT:   t or nil depending upon whether ending provided matches word     *
;                                                                             *
;*****************************************************************************

(define (word-has-ending? word ending)
;#####################################
 (cond
   ((null ending) t)                                  ; no ending? --> matches
   (t
    (and                                              ; both
       (eq (last-char word) (last-char ending)) ;  same last char
       (word-has-ending? (chop-off-last-char word) ;  check rest of ending
                         (chop-off-last-char ending) ;   by recursing
       )
    )
   )
 )
)

(define (chop-off-ending  word ending)
;#####################################
 (cond
   ((null ending) word)                               ; finished ending? --> word
   ((eq (last-char word) (last-char ending))          ; same last char?
    (chop-off-ending  (chop-off-last-char word)       ;  chop off last chars
                      (chop-off-last-char ending)    ;  and recurse
    )
   )
   (t nil)                                            ; otherwise not ending
 )
)

(define (count-noun a-list)
;###########################
 (cond
  ((null a-list) nil)
  (t (or (count? (head a-list))        ; head noun of type count?  (macro)
         (count-noun (tail a-list))    ; otherwise recurse on tail
     )
  )
 )
)

(define (verb-or-count-noun a-list)
;###################################
 (cond
  ((null a-list) nil)
  (t (or (count? (head a-list))              ; head noun of type count? (macro)
         (verbp (head a-list))               ; verb? (uses macro)
         (verb-or-count-noun (tail a-list)) ; otherwise recurse on tail
     )
  )
 )
)

(define (chop-off-last-char word)
;#################################
 (cond
   ((eq (length (explode word)) 1) nil)             ; only one char? --> nil
   (t (implode (reverse (cdr (reverse (explode word))))))
 )
)

(define (add-char word char)
;###########################
 (cond
   ((null word) nil)                                  ; no word? --> nil
   (t (implode (reverse (cons char (reverse (explode word))))))
 )
)

(define (last-char word) (ultimate (explode word)))
;##################################################

(define (ultimate list) (car (reverse list)))          ; first of reverse
```

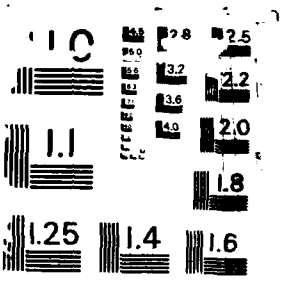
```
;################################################      ;   or first of last

(define (penultimate list) (cadr (reverse list)))      ; second of reverse
;################################################

(define (pen-penultimate list) (caddr (reverse list))) ; third of reverse
;#######################################################

(define (nice-listing word descriptions)
;#########################################
 (eval '(pp ,(cons word descriptions)))
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: look-up                                                         *
;  PURPOSE:  To search the dictionary for an occurance of the given word,    *
;            checking appropriate word roots.                                *
;  INPUT:    A word in "raw" format (ie with any legal suffix or             *
;            conjugation.)                                                   *
;  OUTPUT:   A nice listing of the proper dictionary entry is displayed,     *
;            or an appology is issued.                                       *
;  PROCESS:  The search first begins with known words, followed by an        *
;            examination guided by the endings on words (ie "ing" or "ed").  *
;            The actual dictionary entries reported are built from the       *
;            property lists for the recognized word or word-root.            *
;            The property lists are filtered to assure compatability with    *
;            the word type recognized (ie if the word is plural then change  *
;            the standard entry accordingly)                                 *
;                                                                            *
;*****************************************************************************

(define (look-up word)
;####################
 (let ((filtered-word (filtered-root word)))
  (cond
   ((null filtered-word)                                          ; no entry?
    (tell-user '(I am sorry but I do not know the word ,word) 2 2) ; complain
   )
   (t (nice-listing word filtered-word))                          ; else show
  )
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: look-up-word                                                    *
;  PURPOSE:  To search the dictionary for an occurance of the given word,    *
;            checking appropriate word roots.  A simple version of look-up.  *
;            Returns only a legal a-list or the given word.                  *
;                                                                            *
;*****************************************************************************

(define (look-up-word word)
;##########################
 (let ((filtered-word (filtered-root word)))
  (cond
   ((null filtered-word)                                          ; no entry?
    (tell-user '(I am sorry but I do not know the word ,word) 2 2) ; complain
   )
   (t filtered-word)                                              ; a-list
  )
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: is-a-word                                                       *
;  PURPOSE:  To search the dictionary for an occurance of the given word,    *
;            checking appropriate word roots   A predicate version of lookup*
;                                                                            *
;*****************************************************************************

(define (is-a-word word)
;#######################
 (not (null (filtered-root word)))
)
```

AD-A197 330 A REPORT GENERATOR VOLUME 2(U) AIR FORCE INST OF TECH 2/2
WRIGHT-PATTERSON AFB OH M I MAYBURY 1988
AFIT/CI/NR-88-139

UNCLASSIFIED F/G 12/5 NL



END
DATE
FILMED
9 88



```
(setq *vowel* '(a e i o u y))
(setq *liquid* '(l r s v z))
(setq *noend* '(c g s v z))

(define (n*t-ending? word) (word-has-ending? word 'n*t))
(define (*s-ending? word) (word-has-ending? word '*s))
(define (*-ending? word) (word-has-ending? word '*))
(define (s-ending? word) (word-has-ending? word 's))
(define (ly-ending? word) (word-has-ending? word 'ly))
(define (ing-ending? word) (word-has-ending? word 'ing))
(define (ed-ending? word) (word-has-ending? word 'ed))
(define (en-ending? word) (word-has-ending? word 'en))
(define (er-ending? word) (word-has-ending? word 'er))
(define (est-ending? word) (word-has-ending? word 'est))
(define (ish-ending? word) (word-has-ending? word 'ish))
(define (ize-ending? word) (word-has-ending? word 'ize))

(define (exists? word) (not (null (a-list word))))   ; t if exists, else nil

(define (find word ending)                           ; attempts to find word
 (cond
  ((a-list word) (list word ending (a-list word)))   ; exists --> return all
  (t nil)                                            ; else fail
 )
)
(define (1st word) (last-char word))                 ; last char
(define (2nd word) (1st (chop-off-last-char word)))  ; 2nd from last char
(define (3rd word) (2nd (chop-off-last-char word)))  ; 3rd from last char
(define (1st? word char) (eq (1st word) char))       ; last letter = char?
(define (2nd? word char) (eq (2nd word) char))       ; 2nd to last letter = char?
(define (3rd? word char) (eq (3rd word) char))       ; 3rd to last letter = char?

;*****************************************************************************
;                                                                            *
;  FUNCTION: add                                                             *
;  PURPOSE:  To add or concatenate a given ending to a word.                 *
;                                                                            *
;*****************************************************************************

(define (add word ending)
;########################
 (cond
  ((null word) nil)                                  ; no word -> don't add
  ((null ending) nil)                                ; no ending -> nil
  (t (implode (append (explode word) (explode ending))))); otherwise add it on
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: root                                                            *
;  PURPOSE:  To determine the word type and return the root of the word.     *
;            eventually to return the filtered a-list of the word!           *
;                                                                            *
;  Central ideas from "Procedures as a Representation for Data in a          *
;  Computer Program for Understanding Natural Language," Terry Winograd,    *
;  Ph. D. dissertation, MIT, February 1971.  Cambridge, Massachusetss        *
;  02139.  MIT Project MAC TR-84.                                            *
;                                                                            *
;*****************************************************************************

(define (root word)
;##################
 (cond
  ((a-list word))                                    ; word stored verbatim?
  ((n*t-ending? word)                                ; don*t
   (find (chop-off-ending word 'n*t) 'n*t)           ;  look for it
  )
  ((*s-ending?  word)                                ; Mark*s
   (find (chop-off-ending word '*s) '*s)             ;  look for it
  )
  ((*-ending?   word)                                ; boys*
   (s-root (chop-off-ending word 's*))               ;  test s-ending
  )
  ((s-ending?   word)                                ; lobsters
   (s-root (chop-off-ending word 's))                ;  test s-ending
  )
  ((ly-ending?  word)                                ; slowly
                                                     ; find noun,adj -> verb
   (ly-root (chop-off-ending word 'ly))              ;  test ly-ending
  )
  ((ing-ending? word) (vowel-root (chop-off-ending word 'ing)))  ; blinking
  ((ed-ending?  word)                                ; started
   (vowel-root (chop-off-ending word 'ed)))
  ((en-ending?  word) (vowel-root (chop-off-ending word 'en)))   ; shaken
  ((er-ending?  word) (vowel-root (chop-off-ending word 'er)))   ; faster
```

```
    ((est-ending? word) (vowel-root (chop-off-ending word 'est)))  ; tallest
    ((ish-ending? word)                                  ; sheepish  noun -> adj
     (find (chop-off-ending word 'ish) 'ish)
    )
    (t                                                   ; otherwise express confusion
     (tell-user '(I am sorry but I do not know the word ,word) 2 2)
    )
   )
  )

;*****************************************************************************
;                                                                             *
;  FUNCTION: ly-root                                                          *
;  PURPOSE:  To return the proper ly ending and root of the word, along       *
;            with its unfiltered a-list.                                      *
;                                                                             *
;  NB: word-ly stands for "word minus ly," that is the word stem less the     *
;      ly suffix which is removed prior to this function call.                *
;                                                                             *
;*****************************************************************************


(define (ly-root word-ly)
;########################
 (cond
  ((find                                             ;   (happily -> happy)
     (add (chop-off-ending word-ly 'i) 'y)           ;   add y after chop ily
     'ily                                            ;   "ily" ending
   )                                                 ;   adv --> adj
  )
  (t                                                 ; else  (adv --> verb)
   (cond                                             ;  (slowly -> slow)
    ((find word-ly 'ly))                             ;   try it and if not
    (t                                               ;  else
     (find                                           ;  (cuddly -> cuddle)
       (add word-ly 'le)                             ;   add le to ly-root
       'ly                                           ;   "ly" ending
     )
    )
   )
  )
 )
)

;*****************************************************************************
;                                                                             *
;  FUNCTION: s-root                                                           *
;  PURPOSE:  To return the proper s ending and root of the word, along        *
;            with its unfiltered a-list.                                      *
;                                                                             *
;  NB: word-s stands for "word minus s," that is the word stem less the s     *
;      suffix which is removed prior to this function call.                   *
;                                                                             *
;*****************************************************************************

(define (s-root word-s)
;######################
 (cond
  ((1st? word-s 'e)                                  ; last is e?
   (cond                                             ;
    ((2nd? word-s 'i)                                ; 2nd is i?
     (find (add (chop-off-ending word-s 'ie) 'y) 'ies)  ;  chop ie, add y
    )                                                ;  canaries -> canary
    ((2nd? word-s 'h)                                ; 2nd is h
     (cond
      ((3rd? word-s 't)                              ;  3rd is t?
       (find word-s 's)                              ;   clothes -> clothe
      )                                              ;
      (t                                             ;  else
       (find (chop-off-ending word-s 'e) 'es)         ;   ??hes -> ??h
      )
     )
    )
    ((2nd? word-s 'x)                                ; 2nd is x?
     (find (chop-off-ending word-s 'e) 'es)           ;  boxes -> box
    )
    ((or (2nd? word-s 's) (2nd? word-s 'z))          ; 2nd is s or z?
     (cond
      ((or (3rd? word-s 's) (3rd? word-s 'z))        ;  3rd is s or z?
       (find (chop-off-ending word-s 'e) 'es)         ;   bosses -> boss
      )                                              ;   quizzes -> quiz?
      (t (find word-s 's))                           ;    buses -> bus?
     )                                               ;    fuses -> fuse
    )
    ((2nd? word-s 'v)                                ; 2nd? is v?
     (cond
```

```
      ((find word-s 's))                                    ;  hives -> hive
      ((find (add (chop-off-ending word-s 've) 'f) 'ves)) ;  wolves -> wolf
      ((find (add (chop-off-ending word-s 've) 'fe) 'ves));  knives -> knife
     )
    )
    (t (find word-s 's))                                    ; else look up s
   )
  )
  (t (find word-s 's))                                      ; look up w/out s
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: vowel-root                                                      *
;  PURPOSE:  To return the proper root of the given stem, along with         *
;            with its unfiltered a-list.                                     *
;                                                                            *
;*****************************************************************************

(define (vowel-root stem)
;#########################
 (cond

  ((member (1st stem) *vowel*)                          ; is last char vowel?
   (cond                                                ; yes:
    ((1st? stem 'i)                                     ; last char i?
     (find
       (add (chop-off-ending stem 'i) 'y)               ;  cried -> cry
       'i->y-stem)
     )                                                  ; last char y?
    ((1st? stem 'y)                                     ;  trying -> try
     (find stem 'y-stem)                                ;  played -> play
     )
    ((1st? stem 'e)                                     ; last char e?
     (cond
      ((2nd? stem 'e)                                   ;  2nd to last e?
       (find stem 'ee-stem)                             ;   freeing -> free
       )
      (t
        (cond
          ((find stem '?e-stem))                        ;  else try and if not
          (t (find (add stem 'e) '?ee-stem))            ;   add e and try
         )
       )
      )
     )
    (t (find (add stem 'e) 'aou+e-stem))                ; else add e and look
   )
  ) ; close 1st VOWEL

  ((1st? stem 'h)                                     ; last char h?
   (cond
    ((2nd? stem 't)                                   ; 2nd to last t?
     (cond
      ((find stem 'th-stem))                          ;  look for th
      (t (find (add stem 'e) 'the-stem))              ;  else add e
     )
    )
    (t (find stem 'th-stem))                          ; else add e
   )
  )

  ((eq (1st stem) (2nd stem))                         ; last = 2nd to last?
   (cond
    ((member (1st stem) *liquid*)                     ;  last in *liquid*?
     (cond                                                 kissing -> kiss
      ((find stem 'XX-stem))                          ;   try fizzing -> fizz
      ((find (chop-off-last-char stem) 'XX->X-stem)) ;   but quizzing -> quiz
     )
    )
    (t (find (chop-off-last-char stem) 'XX->X-stem2));  else try w/out last
   )
  )

  ((member (2nd stem) *vowel*)
   (cond
    ((member (3rd stem) *vowel*)
     (cond
      ((member (1st stem) *noend*)
       (find (add stem 'e) 'VVN->VVNe-stem))
      ((find stem 'VVN-stem))
     )                                                ; gaming -> game
    )                                                 ; liking -> like
    (t                                                ; modified from T. W.
     (cond                                            ; add e to the stem
```

```
        ((find (add stem 'e) '->e-stem))                ; liked -> like
        ((find stem 'stem))                             ; suffered -> suffer
       )
      )
     )
    )

    ((member (1st stem) *liquid*)                       ; is last in liquid?
     (cond
      ((chop-off-ending stem 'rl)                       ; if the ending is rl
       (find stem 'rl-stem))                            ;  look it up
      (t (find (add stem 'e) 'L-stem))                  ; else add an e
     )
    )

    (t                                                  ; else
     (cond
      ((member (1st stem) *noend*)                      ; last char in noend?
       (find (add stem 'e) '->e-stem2))                 ;  add an e
      (t (find stem 'stem))                             ; else look up plain
     )
    )
   )
  )

  (define (add-verb-3ps a-list)
   (cond
    ((null a-list) nil)
    (t
     (let ((defn (first a-list)))
      (cons
       (list
        (append (list (first (syntax defn)) 'sing3p)
                (nthcdr 3 (syntax defn)))
        (semantics defn)
        (realization defn))
       a-list
      )
     )
    )
   )
  )

  (define (delete-word-verb a-list)
   (cond
    ((null a-list) nil)
    (t
     (cons
      (list (my-delete 'verb (syntax (head a-list)))
            (semantics (head a-list))
            (realization (head a-list)))
      (delete-word-verb (tail a-list))
     )
    )
   )
  )

  ;*****************************************************************************
  ;                                                                             *
  ;  FUNCTION: filtered-root                                                    *
  ;  PURPOSE:  An improvment upon root since here a-lists are filtered and      *
  ;            adjusted according to the mophology of the input.  Thus          *
  ;            adjustments in word type, word tense, or word number are         *
  ;            made where appropriate.                                          *
  ;                                                                             *
  ;*****************************************************************************

  (define (filtered-root word)
  ;############################
   (cond
    ((delete-word-verb (a-list word)))                  ; word stored verbatim?
    ((n't-ending? word)                                 ; don't
      (make-verbs-n't
       (third (find (chop-off-ending word 'n't) 'n't))));  look for it
    )
    ((*s-ending?  word)                                 ; Mark*s
       (make-nouns-and-names-singular-possessive        ;  singular possessive
        (third (find (chop-off-ending word '*s) '*s))) ;  look for it
    )
    ((*-ending?   word)                                 ; boys*
       (make-nouns-plural-possessive                    ;  plural possessive
        (third (s-root (chop-off-ending word 's*))))    ;  test s-ending
    )
    ((s-ending?   word)                                 ; lobsters
       (make-nouns-plural-and-verbs-3ps
         (third (s-root (chop-off-ending word 's))))    ;  test s-ending
```

```
    )
    ((ly-ending? word)                                  ; slowly
        (make-adjectives-and-nouns-adverbs              ;  adj -> adverbs
         (third (ly-root (chop-off-ending word 'ly)))) ;  test ly-ending
    )
    ((ing-ending? word)                                 ; blinking
       (make-verbs-ing
        (third (vowel-root (chop-off-ending word 'ing))))
    )
    ((ed-ending? word)                                  ; started
       (add-verb-3ps
        (make-verbs-past-tense                          ; past tense
         (third (vowel-root (chop-off-ending word 'ed)))))
    )
    ((en-ending? word)                                  ; shaken
        (make-verbs-past-participle                     ; past participle
         (third (vowel-root (chop-off-ending word 'en))))
    )
    ((er-ending? word)                                  ; faster
        (make-adjectives-comparative                    ; comparative adjective
          (third (vowel-root (chop-off-ending word 'er))))
    )
    ((est-ending? word)                                 ; tallest
        (make-adjectives-superlative                    ; superlative adjective
         (third (vowel-root (chop-off-ending word 'est))))
    )
    ((ish-ending? word)                                 ; sheepish  noun -> adj
        (make-nouns-adjectives                          ; adjective
         (third (find (chop-off-ending word 'ish))))
    )
    (t nil)                                             ; otherwise express confusion
   )
  )
```

```
;*****************************************************************************
;                                                                             *
;  MODULE:   GRAMMAR SUPPORT FOR FEATURES                                     *
;  PURPOSE:  To support the grammar accessing functions to allow features     *
;            in the grammar.                                                  *
;                                                                             *
;*****************************************************************************


;*****************************************************************************
;                                                                             *
;  FUNCTION: left-hand-side-of                                                *
;  PURPOSE:  To left the right hand side of a give rule.                      *
;                                                                             *
;*****************************************************************************

(define (left-hand-side-of rule) (head rule))
;###############################################

;*****************************************************************************
;                                                                             *
;  FUNCTION: right-hand-side-of                                               *
;  PURPOSE:  To return the right hand side of a give rule.                    *
;                                                                             *
;*****************************************************************************

(define (right-hand-side-of rule) (tail rule))
;###############################################

;*****************************************************************************
;                                                                             *
;  FUNCTION: is-a-non-terminal                                                *
;  PURPOSE:  To determine if a given symbol in the grammar is a non-          *
;            terminal (ie it is on the left hand side of a grammar rule)      *
;                                                                             *
;*****************************************************************************

(define (is-a-non-terminal symbol grammar)
;##########################################
 (cond
    ((null grammar) nil)                                  ; finished?  - quit
    ((equal symbol                                        ; match?
      (first (left-hand-side-of                           ; first item lhs
       (rule-syntax (head grammar))))) t)                 ;    -> succeed
    (t (is-a-non-terminal symbol (tail grammar)))         ; else recurse
 )
)

;*****************************************************************************
;                                                                             *
;  FUNCTION: is-a-terminal                                                    *
;  PURPOSE:  To determine if the given symbol is a terminal.                  *
;                                                                             *
;*****************************************************************************

(define (is-a-terminal symbol grammar)
;#####################################
  (not (is-a-non-terminal symbol grammar))
)

;*****************************************************************************
;                                                                             *
;  FUNCTION: a-rule-to-expand                                                 *
;  PURPOSE:  To randomly choose a rule from all the rules expanding a         *
;            symbol in the given grammar.                                     *
;                                                                             *
;*****************************************************************************

(define (a-rule-to-expand symbol grammar)
;########################################
 (pick-at-random-from (all-rules-expanding symbol grammar))
)

;*****************************************************************************
;                                                                             *
;  FUNCTION: pick-at-random-from                                              *
;  PURPOSE:  To pick an item randomly from a given list.                      *
;                                                                             *
;*****************************************************************************

(define (pick-at-random-from list)
;#################################
  (nthelem (my-random (length list)) list)
)

;*****************************************************************************
```

```
;                                                                              *
;  FUNCTION: all-rules-expanding                                               *
;  PURPOSE:  To return all rules in the grammar that expand given symbol       *
;                                                                              *
;*******************************************************************************

(define (all-rules-expanding symbol grammar)
;############################################
 (cond
  ((null grammar) nil)
  ((equal symbol (first (rule-syntax (head grammar))))
   (cons (head grammar)
         (all-rules-expanding symbol (tail grammar))))
  (t (all-rules-expanding symbol (tail grammar)))
 )
)

;*******************************************************************************
;                                                                              *
;  FUNCTION: expand-all-symbols                                                *
;  PURPOSE:  To convert a list of symbols into their right hand side of        *
;            the rule equivalent (ie expand them).                             *
;                                                                              *
;*******************************************************************************

(define (expand-all-symbols list grammar)
;########################################
 (cond
  ((null list) nil)
  ((is-a-terminal (car list) grammar)                ; terminal?
    (cons (car list)                                 ; add it to
          (expand-all-symbols (cdr list) grammar)    ; result of recurse
    )
  )
  (t (append                                         ; else make list of
       (right-hand-side-of                           ; expansion symbol
          (rule-syntax (a-rule-to-expand (car list) grammar)))
       (expand-all-symbols (cdr list) grammar)       ; and recursion
     )
  )
 )
)

;*******************************************************************************
;                                                                              *
;  FUNCTION: generate-sentence                                                 *
;  PURPOSE:  To generate a sentence from the given grammar.                    *
;                                                                              *
;*******************************************************************************

(define (generate-sentence grammar)
;##################################
 (print-list
  (generate-phrase
    (generate-phrase-structures '(s) nil grammar)
  )
 )
)

;*******************************************************************************
;                                                                              *
;  FUNCTION: generate-phrase                                                   *
;  PURPOSE:  To take a phrase structure (list) and recursively find            *
;            examples of each member in the phrase, consing together to        *
;            return a list result.                                             *
;                                                                              *
;*******************************************************************************

(define (generate-phrase phrase-structure)
;##########################################
 (cond
   ((null phrase-structure) nil)                  ; no more phrases? --> nil
   (t (cons                                       ; otherwise attach example
        (find-example-of (head phrase-structure)) ;  of head category
        (generate-phrase (tail phrase-structure)) ;  to tail recurse
      )
   )
 )
)

;*******************************************************************************
;                                                                              *
;  FUNCTION: generate-phrase-structures                                        *
;  PURPOSE:  To generate phrase structures from a phrase list by               *
;            expanding all symbols in the phrase list until you can't          *
;            expand anymore.                                                   *
```

```
;
;*****************************************************************************

(define (generate-phrase-structures phrase-list last-time grammar)
;#############################################################
 (cond
  ((equal phrase-list last-time) phrase-list)
  (t (msg phrase-list N)                        ; if this included, prints.
     (generate-phrase-structures
       (expand-all-symbols phrase-list grammar) phrase-list grammar)
  )
 )
)
```

```
;////////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                               |
;  MODULE:  LEXICAL CATEGORY SEARCH AND MATCH                                   |
;  PURPOSE: To search and match categories within a word description.           |
;                                                                               |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////////

;*******************************************************************************
;                                                                              *
;  FUNCTION: lexical-category                                                  *
;  PURPOSE:  To determine the lexical category for a given description.        *
;  INPUT:    description for some word                                         *
;  OUTPUT:   lexical category of description                                   *
;                                                                              *
;*******************************************************************************

(define (lexical-category description)
;##########################
 (cond
  ((eq (word-type description) 'verb)          ; if word type is verb
   (verb-type description))                    ;  return verb type
  (t (word-type description))                  ; else give word type
 )
)

(define (lexical-category-and-semantics description)
;##################################################
 (cond
  ((eq (word-type description) 'verb)          ; if word type is verb
   (append
    (list (verb-type description))             ;  return verb type and semantics
    (semantics description)))
  (t
   (append
    (list (word-type description))             ; else give word type
    (semantics description)))
 )
)

;*******************************************************************************
;                                                                              *
;  FUNCTION: all-lexical-categories                                            *
;  PURPOSE:  To compute a list of all lexical categories in the given a-list   *
;  INPUT:    a-list for some word                                              *
;  OUTPUT:   a list of all lexical categories in a-list or nil                 *
;                                                                              *
;*******************************************************************************

(define (all-lexical-categories a-list)
;######################################
 (cond
  ((null a-list) nil)                          ; no more descriptions?
  (t (cons                                     ; make a list of
       (lexical-category (head a-list))        ;  lex cat of 1st description
       (all-lexical-categories (tail a-list)); lex cats of rest (recurse)
     )
  )
 )
)

(define (all-lexical-categories-and-semantics a-list)
;####################################################
 (cond
  ((null a-list) nil)                                    ; no more descriptions?
  (t (cons                                               ; make a list of
       (lexical-category-and-semantics (head a-list)) ;  lex cat of 1st description
       (all-lexical-categories-and-semantics (tail a-list)); lex cats of rest (recurse)
     )
  )
 )
)

;*******************************************************************************
;                                                                              *
;  FUNCTION: lexical-categories                                                *
;  PURPOSE:  To determine the list of lexical categories a word belongs to.    *
;            First looks up the word in the dictionary, checking alternate     *
;            spellings and roots.                                              *
;  INPUT:    a word in the dictionary                                          *
;  OUTPUT:   list of lexical categories in a-list or nil                       *
;  OPTIONS:  1 -- return only list of lexical categories                       *
;            2 -- return list of categories with semantics.                    *
;                                                                              *
;*******************************************************************************
```

```
(define (lexical-categories word)
;################################
 (delete-duplicates (all-lexical-categories (look-up-word word)))
)

(define (lex-cats-and-semantics word)
;#####################################
 (delete-duplicates
   (all-lexical-categories-and-semantics
     (look-up-word word)))
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: current-word                                                    *
;  PURPOSE:  To return the lexical-categories of the next word in the input.*
;                                                                            *
;*****************************************************************************

(define (current-word sentence)
;##############################
  (delete-duplicates (lexical-categories (head sentence)))
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: rest-of                                                         *
;  PURPOSE:  To return the rest of the given sentence after stripping off    *
;            first word.                                                     *
;                                                                            *
;*****************************************************************************

(define (rest-of sentence) (tail sentence))
;###########################################

;*****************************************************************************
;                                                                            *
;  FUNCTION: end-of-sentence?                                                *
;  PURPOSE:  To return t if the sentence is empty, otherwise nil.            *
;                                                                            *
;*****************************************************************************

(define (end-of-sentence? sentence)
;###################################
  (cond ((null sentence) t)                          ; sentence empty? --> yes!
        (t nil)                                      ; otherwise nil
  )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: word-or-lexical-category?                                       *
;  PURPOSE:  To return t if the item is a word or lexical category, else nil*
;                                                                            *
;*****************************************************************************

(define (word-or-lexical-category? item)
;########################################
 (cond
  ((or (is-a-category item) (is-a-word item)) t)  ; t if lexical cat or word
  (t nil)                                         ; otherwise nil
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: match                                                           *
;  PURPOSE:  To return t if item1 is in the lexical cat of item 2.           *
;                                                                            *
;*****************************************************************************

(define (match item1 item2)
;###########################
 (or (eq item1 item2)                                    ; how about look-up???
     (member                                             ; member?
         item1                                           ;  item1
         (all-lexical-categories (a-list (root item2))));  lex cats of item2
     )
 )
)
```

preprocess.l Wed Aug 19 05:13:17 1987

```
;////////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                               |
;  MODULE:    PREPROCESS                                                        |
;  PURPOSE:   To preprocess the feature grammar for efficiency.                 |
;  OWNER:     copywrite Mark T. Maybury, August, 1987.                          |
;                                                                               |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////////

;-------------------------------------------------------------------------------
; FUNCTION: preprocess                                                          |
; PURPOSE:  to preprocess the grammar.                                          |
;-------------------------------------------------------------------------------

(define (preprocess grammar)
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 (let ((grammar (compile-grammar grammar)))
  (make-possible-rules grammar grammar)      ; find possible rules
  (make-rules-with-handle grammar)           ; find rules with handle
  (make-rules-starting-with grammar)
 )
)

;-------------------------------------------------------------------------------
; FUNCTION:  make-rules-starting-with                                           |
; PURPOSE:   to provide an index into the grammar based on rhs rule.            |
;-------------------------------------------------------------------------------

(define (make-rules-starting-with grammar)
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 (cond
  ((null grammar) nil)
  (t
   (let* ((rule (head grammar))
          (constituent (caaar (tail rule))))
    (putprop
      constituent
      (cons rule (rules-starting constituent))
      'starting)
    (make-rules-starting-with (tail grammar))
   )
  )
 )
)

(define (rules-starting category) (get category 'starting))
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
;////////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                               |
;  MODULE:  COMPILE GRAMMAR MODULE                                              |
;  PURPOSE: To provide tools to compile the grammar prior to execution to       |
;           give computational efficiency. This allows for grammatical          |
;           felicity or perspicuity when writing rules.                         |
;                                                                               |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////////

; Grammar originally of form:

; grammar     ==> rule*
; rule        ==> ( rule-name rule-syntax rule-semantics )
; rule-syntax ==> constituent*
; constituent ==> ( category (feature value)* )

; compiling performs:
; 1. removes feature so unification becomes positional
;    this is done for both computational efficiency as well as
;    convenience for use with dictionary.

;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////////


;-------------------------------------------------------------------------------
; FUNCTION: compile-grammar                                                     |
; PURPOSE:  to replace the global variable *grammar* with the compiled form.    |
;-------------------------------------------------------------------------------

(define (compile-grammar grammar)
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 (setq *grammar* (mapcar 'remove-lables *grammar*)))

;-------------------------------------------------------------------------------
; FUNCTION: remove-lables                                                       |
; PURPOSE:  to remove the feature lables on the syntax of a rule.               |
;-------------------------------------------------------------------------------

(define (remove-lables rule)
;~~~~~~~~~~~~~~~~~~~~~~~~~~~
 (list
  (rule-name rule)
  (mapcar 'rem-feature-lables (rule-syntax rule))
  (rule-semantics rule)
 )
)

;-------------------------------------------------------------------------------
; FUNCTION: rem-feature-lables                                                  |
; PURPOSE:  to remove the feature lables on a particular constituent.           |
;-------------------------------------------------------------------------------

(define (rem-feature-lables constituent)
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 (cons
  (head constituent)
  (mapcar 'cadr (tail constituent))
 )
)
```

```
;////////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                               |
;  MODULE:  PREPROCESS                                                          |
;  PURPOSE: To preprocess the grammar to increase parser efficiency.            |
;  $Header: preprocess.l,v 1.1 86/12/15 01:11:06 mtm Exp $                      |
;                                                                               |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////////

;*******************************************************************************
;                                                                              *
;  FUNCTION: make-possible-rules                                               *
;  PURPOSE:  To associate the possible rules for a particular lexical          *
;            category.                                                         *
;  INPUT:    grammar                                                           *
;  OUTPUT:   Nothing explicitly, but side effect is possible rules for cats.   *
;                                                                              *
;*******************************************************************************

(define (make-possible-rules grammar syntax-rules)
;#################################################
  (clear-possible-rules grammar)
  (create-possible-rules grammar grammar)
)

;*******************************************************************************
;                                                                              *
;  FUNCTION: clear-possible-rules                                              *
;  PURPOSE:  To clear the lexical categories of their property                 *
;            "possible-rules", which may be dirty from previous runs.          *
;            Maps the function clear-category on the rhs of each rule.         *
;                                                                              *
;*******************************************************************************

(define (clear-possible-rules rest-of-grammar)
;#############################################
 (cond
  ((null rest-of-grammar) nil)
  (t (mapc 'clear-category                                    ; clear
      (mapcar 'car                                            ; the first
        (right-hand-side-of (rule-syntax (head rest-of-grammar)))))  ; of items in rhs
     (clear-possible-rules (tail rest-of-grammar))            ; fix tail
  )
 )
)

(define (clear-category symbol) ; don't waste time checking if a category
;###############################
 (setplist symbol nil)
)

;*******************************************************************************
;                                                                              *
;  FUNCTION: create-possible-rules                                             *
;  PURPOSE:  To assign to each lexical category its possible rules.            *
;                                                                              *
;*******************************************************************************

(define (create-possible-rules grammar-rules grammar)
;####################################################
 (cond
  ((null grammar-rules) nil)                          ; no more rules? -> quit
  (t                                                  ; else
   (make-rule-possible-for-lexical-categories         ;  make possible for firsts
     (head grammar-rules)                             ;   this rule
     (first2 (first (first (right-hand-side-of        ;   first2 of category
               (rule-syntax (head grammar-rules)))))) ;     of its handle
             grammar
     )
   )
   (create-possible-rules                             ;  recurse on rest
      (tail grammar-rules)
      grammar
   )
  )
 )
)

;*******************************************************************************
;                                                                              *
;  FUNCTION: make-rule-possible-for-lexical-categories                         *
;  PURPOSE:  To make the given rule a possible rule for all the lexical        *
;            categories in firsts.                                             *
;                                                                              *
;*******************************************************************************
```

```
(define (make-rule-possible-for-lexical-categories rule firsts)
;###############################################################
 (cond
  ((null firsts) nil)                              ; finished? -> quit
  (t (cond                                         ; else
      ((is-a-category (head firsts))               ;  head firsts lexical category?
       (add-to-possible-rules rule (head firsts))) ;  then rule is possible for
      )
     )
     (make-rule-possible-for-lexical-categories    ;  recurse on rest
        rule
        (tail firsts)
     )
  )
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: add-to-possible-rules                                           *
;  PURPOSE:  To add the give rule to the provided category's possible rules. *
;                                                                            *
;*****************************************************************************

(define (add-to-possible-rules rule category)
;############################################
 (putprop
    category
    (cons rule (possible-rules category))        ; cons it to old possible rules
    'possible-rules
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: possible-rules                                                  *
;  PURPOSE:  To return the rules which a possible from the given category.   *
;                                                                            *
;*****************************************************************************

(define (possible-rules category) (get category 'possible-rules))
;################################################################

;*****************************************************************************
;                                                                            *
;  FUNCTION: display-category                                                *
;  PURPOSE:  To display the possible rules for this category, as well as     *
;            rules with this category as a handle (a handle is the first     *
;            constituent on the right hand side of a rule).                  *
;                                                                            *
;*****************************************************************************

(define (display-category category)
;##################################
 (blank 2)
 (print-list '(---- ,category ----)) (blank 2)
 (print-list '(possible rules ----> ,@(possible-rules category))) (terpri)
 (print-list '(rules with handle -> ,@(rules-with-handle category)))
 (blank 2)
)


;*****************************************************************************
;                                                                            *
;  FUNCTION: First2                                                          *
;  PURPOSE:  To return a list (containing no duplicates) of:                 *
;                                                                            *
;                the result of first-aux1                                    *
;              + all non-nil instances of                                    *
;                    first-aux2 for the rhs of all rules in the grammar      *
;                    where category equals the lhs of the rule               *
;                                                                            *
;  INPUT:    A category and a Grammar                                        *
;  OUTPUT:   list                                                            *
;                                                                            *
;*****************************************************************************

(define (first2 category grammar)
;################################
 (delete-duplicates
   (append (first-aux1 category grammar nil)
           (first-aux2-for-lhs-matching-rules category grammar)
   )
 )
)
```

```
;*****************************************************************************
;                                                                            *
;  FUNCTION: first-aux2-for-lhs-matching-rules                               *
;  PURPOSE:  To return a list of all non nil instances of doing:             *
;              for each rule in the grammar (recurse)                        *
;              if the category given equals the lhs of the rule              *
;                 then return the result of first-aux2 on the rhs of rule    *
;                                                                            *
;*****************************************************************************

(define (first-aux2-for-lhs-matching-rules category rest-of-grammar)
;#############################################################################
 (cond
  ((null rest-of-grammar) nil)                                  ; finished? -> stop
  ((eq category
       (left-hand-side-of (rule-syntax (head rest-of-grammar)))) ; cat = lhs rule?
   (append                                                      ;  add
      (first-aux2                                               ;   first-aux2
          (right-hand-side-of (rule-syntax (head rest-of-grammar))) ; on next rule
          grammar)
      (first-aux2-for-lhs-matching-rules                        ;   tail recurse
          category
          (tail rest-of-grammar))
   )
  )                                                             ; else tail recurse
  (t (first-aux2-for-lhs-matching-rules category (tail rest-of-grammar)))
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: first-aux1                                                      *
;  PURPOSE:  To return:                                                      *
;                                                                            *
;  If the category is terminal then return a list containing the category    *
;  otherwise, if the category has already been tried the return nil,         *
;  otherwise return a list of all non-nil instance of the following:         *
;  check each rule in the grammar:                                           *
;    if the category is equal to the left hand side of the rule              *
;       then if the right hand side of the rule is not empty                 *
;         then return the result of                                          *
;          first-aux1 of the first symbol in the rhs of the rule             *
;          where you add the current category to already tried.              *
;       else return a list containing the "empty" symbol                     *
;    else return nil                                                         *
;                                                                            *
;*****************************************************************************

(define (first-aux1 category grammar already-tried)
;##################################################
 (cond
  ((is-a-terminal category grammar) (list category))
  ((member category already-tried) nil)
  (t (first-aux1-lhs-matches-category category grammar))
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: first-aux1-lhs-matches-category                                 *
;  PURPOSE:  To return a list of all non nil instances of doing:             *
;     check each rule in the grammar:                                        *
;       if the category is equal to the left hand side of the rule           *
;          then if the right hand side of the rule is not empty              *
;            then return the result of                                       *
;             first-aux1 of the first symbol in the rhs of the rule          *
;             where you add the current category to already tried.           *
;          else return a list containing the "empty" symbol                  *
;       else return nil                                                      *
;                                                                            *
;*****************************************************************************

(define (first-aux1-lhs-matches-category category rest-of-grammar)
;#################################################################
 (cond
  ((null rest-of-grammar) nil)                                  ; finished? -> stop
  ((eq category
       (left-hand-side-of (rule-syntax (head rest-of-grammar)))) ; cat = lhs rule?
   (cond
    ((not (null (right-hand-side-of
                  (rule-syntax (head rest-of-grammar)))))   ; rhs rule not empty?
     (append
      (first-aux1                                           ;   first-aux1
          (first (right-hand-side-of
                   (rule-syntax (head rest-of-grammar))))   ;    1st on next rule
```

```
            grammar
            (append1 already-tried category))
       (first-aux1-lhs-matches-category                    ; and tail recurse
            category
           (tail rest-of-grammar))
       )
      )
      (t (append
           (list (list 'empty-symbol))                    ; else empty-symbol
           (first-aux1-lhs-matches-category               ; and tail recurse
             category
             (tail rest-of-grammar))
         )
       )
      )
     )
     (t (first-aux1-lhs-matches-category                  ;else tail recurse
          category
          (tail rest-of-grammar)
        )
     )
    )
   )
  )

;*************************************************************************
;                                                                        *
;  FUNCTION: first-aux2                                                  *
;  PURPOSE:  To take away some computational burden of first.            *
;                                                                        *
;  If rhs is empty, return empty list.                                   *
;  Otherwise                                                             *
;    if first-aux1 of the first symbol in rhs with tried nil, has "empty"*
;    then return list containing:                                        *
;       first-aux1 of the first symbol in rhs with already-tried nil     *
;     + first-aux2 of all but first symbol in rhs                        *
;    else first-aux1 of the first symbol in rhs with alread-tried nil.   *
;                                                                        *
;*************************************************************************

(define (first-aux2 rhs grammar)
;###############################
 (cond
  ((null rhs) nil)                                     ; rhs empty -> empty list
  ((member 'empty-symbol                               ; contains empty-symbol
           (first-aux1 (first rhs) grammar nil))       ; first-aux1?
   (append
     (first-aux1 (first rhs) grammar nil)              ; first-aux1 head +
     (first-aux2 (tail rhs) grammar)                   ; first-aux2 tail
   )
  )
  (t (first-aux1 (first rhs) grammar nil))             ; else first-aux1 head
 )
)

;#########################################################################
;#######################    HANDLES    ###################################
;#########################################################################

; grammar assumed to include features in the form:

; ( (r1 [nt f v ... f v]  [s1 f v ... f v] ... [sn f v ... f v])
;
;    .
;    .
;   (rn [nt f v ... f v]  [sx f v ... f v] ... [sy f v ... f v])
; )
;
; where nt = non-terminal, f = feature, v = feature value, si = symbol i
;       ri = rule i
;

;*************************************************************************
;                                                                        *
;  FUNCTION: make-handle-rules                                           *
;  PURPOSE:  To place the rules with this handle on the p-list of handle.*
;                                                                        *
;*************************************************************************

(define (add-handle-rules handle rules-with-handle)
;##################################################
 (putprop
    handle
    rules-with-handle
    'rules-with-handle
 )
)
```

```
;*****************************************************************************
;                                                                            *
;  FUNCTION: make-rules-with-handle                                          *
;  PURPOSE:  To index rules by their handle in the given grammar.            *
;                                                                            *
;*****************************************************************************

(define (make-rules-with-handle grammar)
;#######################################
 (make-rules-with-handle2 (all-handles grammar) grammar)
)

(define (make-rules-with-handle2 handles grammar)
;################################################
 (cond
  ((null handles) nil)
  (t (add-handle-rules
       (head handles)
       (all-rules-with-handle (head handles) grammar))
     (make-rules-with-handle2 (tail handles) grammar)
  )
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: rules-with-handle                                               *
;  PURPOSE:  To return the rules which have the given category as the        *
;            first constituent in the right hand side of the rule.           *
;                                                                            *
;*****************************************************************************

(define (rules-with-handle category) (get category 'rules-with-handle))
;#######################################################################

;*****************************************************************************
;                                                                            *
;  FUNCTION: all-handles                                                     *
;  PURPOSE:  To return all non-duplicate first constituents on the right     *
;            hand side of the rules in the grammar.                          *
;                                                                            *
;*****************************************************************************

(define (all-handles grammar) (delete-duplicates (mapcar 'caadadr grammar)))
;###########################################################################

(define (all-feature-handles grammar)
  (delete-duplicates (all-feature-handles2 grammar)))

(define (all-feature-handles2 grammar)
 (cond
  ((null grammar) nil)                              ; first feature variable?
  ((variable? (second (second (rule-syntax (head grammar)))))
   (cons                                            ; add
    (first (second (rule-syntax (head grammar))))   ; the cat of 1st on rhs
    (all-feature-handles2 (tail grammar))           ;  to rest of handles
   )
  )
  (t                                                ; else
   (cons                                            ;  add
    (second (second (rule-syntax (head grammar))));  first feature constant
    (all-feature-handles2 (tail grammar))           ;   to rest of handles
   )
  )
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: all-rules-with-handle                                           *
;  PURPOSE:  To return all rules in the grammar that have the given symbol   *
;            in the first position of the rhs of the rule.                   *
;                                                                            *
;                                                                            *
;*****************************************************************************

(define (all-rules-with-handle symbol grammar)
;#############################################
 (cond
  ((null grammar) nil)
  ((equal
     symbol                                         ; is symbol the first
     (first (second (rule-syntax (head grammar))))) ; category on rhs rule?
   (cons (head grammar)
         (all-rules-with-handle symbol (tail grammar))))
```

```
    (t (all-rules-with-handle symbol (tail grammar)))
  )
)
```

```
;////////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                               |
;  MODULE:  VERTEX_EDGE                                                         |
;  PURPOSE: To provide vertex/edge accessing and alteration capabilities.       |
;                                                                               |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////////

;*******************************************************************************
;                                                                               *
;  FUNCTION: add-vertex-to-chart                                                *
;  PURPOSE:  To add given vertex to the chart which is a global variable.       *
;                                                                               *
;*******************************************************************************

(define (add-vertex-to-chart vertex chart)
;##########################################
  (setq *chart* (append1 *chart* vertex))
)

;###############################################################################
;###############################################################################
;#############          VERTEX/EDGE ACCESSING FUNCTIONS             ############
;###############################################################################
;###############################################################################
;  values stored on p-lists

;=============================================================================
; edge has the form < left-vertex, right-vertex, category, needed, contents >
;=============================================================================

(defmacro left-vertex-of  (edge) '(get ,edge 'left-vertex))
(defmacro right-vertex-of (edge) '(get ,edge 'right-vertex))
(defmacro category-of     (edge) '(get ,edge 'category))
(defmacro needed-of       (edge) '(get ,edge 'needed))
(defmacro contents-of     (edge) '(get ,edge 'contents))
(defmacro rule-of         (edge) '(get ,edge 'rule))

(define (whole-edge edge)
;########################
 (list (left-vertex-of edge) (right-vertex-of edge) (category-of edge)
       (needed-of edge) (contents-of edge) (rule-of edge))
)

; nice printout of display

(define (display-edge edge)
;##########################
 (blank 2)
 (print-list (list '---- edge '----)) (blank 2)
 (print-list '(left vertex ---> ,(left-vertex-of edge)))  (terpri)
 (print-list '(right vertex --> ,(right-vertex-of edge))) (terpri)
 (print-list '(category ------> ,(category-of edge)))     (terpri)
 (print-list '(needs ---------> ,(needed-of edge)))       (terpri)
 (print-list '(contents ------> ,(contents-of edge)))     (terpri)
 (print-list '(rule applied --> ,(rule-of edge)))         (terpri)
 (blank 1)
)

(define (incomplete? edge) (needed-of edge))       ; if needed of edge is
;###############################################       ;  non nil then incomplete


(define (complete? edge) (not (incomplete? edge))) ; if needed of edge is
;##################################################
   ; equivalent to -->   (not (needed-of  edge)))  ;    nil then finished


;=============================================================================
; vertex has the form < edges-in, edges-out >                                  |
;=============================================================================

(defmacro edges-in  (vertex) '(get ,vertex 'edges-in))
(defmacro edges-out (vertex) '(get ,vertex 'edges-out))
(defmacro whole-vertex (vertex) '(list (edges-in ,vertex) (edges-out ,vertex)))

;*******************************************************************************
;                                                                               *
;  FUNCTION: display-vertex-td                                                  *
;  PURPOSE:  To display the vertex on the chart for a top down parse.           *
;                                                                               *
;*******************************************************************************

(define (display-vertex-td vertex)
;#################################
```

```
 (blank 2)
 (print-list '(---- ,vertex ----)) (blank 2)
 (print-list '(edges in ---> ,@(edges-in vertex)))  (terpri)
 (print-list '(edges out --> ,@(edges-out vertex)))
 (print-list '(proposed ---> ,@(category-list (edges-out vertex))))
 (blank 2)
)

(define (category list edge-list)         ; returns the categories in edge list
;################################
 (mapcar 'category-of edge-list))

;*****************************************************************************
;                                                                            *
;  FUNCTION: display-vertex-bu                                               *
;  PURPOSE:  To display the vertex on the chart for a bottom up parse.       *
;                                                                            *
;*****************************************************************************

(define (display-vertex-bu vertex)         ; NB diverse from top down version
;################################
 (blank 2)
 (print-list '(---- ,vertex ----)) (blank 2)
 (print-list '(edges in ---> ,@(edges-in vertex)))  (terpri)
 (print-list '(edges out --> ,@(edges-out vertex))) (terpri)
 (print-list '(rules proposed ---> ,@(rules-proposed-at vertex))) (terpri)
 (print-list '(real rules proposed --> ,@(real-rules-proposed-at vertex)))
 (blank 2)
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: rules-proposed-at/real-rules-proposed-at                        *
;  PURPOSE:  To return the rules which have been proposed at the vertex      *
;            or to return the real rules proposed (ie with actual varaibles  *
;            used).                                                          *
;*****************************************************************************

(define (rules-proposed-at vertex) (get vertex 'rules-proposed))
;###############################################################

(define (real-rules-proposed-at vertex) (get vertex 'real-rules-proposed))
;#########################################################################



;#########################################################################################
;#########################################################################################
;############          VERTEX/EDGE ALTERATION FUNCTIONS                 #########
;#########################################################################################
;#########################################################################################

;=========================================================================

(define (add-edge-to-chart edge chart track-level)
;#################################################
 (add-edge-out edge (left-vertex-of edge) track-level)
 (add-edge-in  edge (right-vertex-of edge) track-level)
)

(define (add-edge-out edge vertex track-level)
;#############################################
   (putprop
     vertex
     (append1 (edges-out vertex) edge)          ; add edge to end of edges-out
     'edges-out
   )
)

(define (add-edge-in edge vertex track-level)
;############################################
   (putprop
     vertex
     (append1 (edges-in vertex) edge)           ; add edge to end of edges-in
     'edges-in
   )
)

;=========================================================================



;*****************************************************************************
;                                                                            *
;  FUNCTION: new-vertex                                                      *
;  PURPOSE:  To create a new identifier for the vertex.  Use built-in        *
```

```
;            function newsym which returns new symbols with incrementally   *
;            higher postscripts (ie it generate vertex0, then vertex1, etc.)*
;                                                                            *
;*****************************************************************************

(defmacro new-vertex ()
;#######################
   '(clear-vertex (newsym 'vertex)))

(define (clear-vertex vertex)
;############################
 (setplist vertex nil)
  vertex                                        ; return name
)

(defmacro new-vertex2 ()                        ; alterate form of new vertex
;#######################
'(progn
  (let ((new-vertex (newsym 'vertex)))
    (setplist new-vertex nil)
    new-vertex
  )
 )
)


;*****************************************************************************
;                                                                            *
;  FUNCTION: create-edge                                                     *
;  PURPOSE:  To create a new identifier for the edge, and associate the      *
;            components with it.                                             *
;                                                                            *
;  ARGUMENTS: < left-vertex, right-vertex, category, needed, contents        *
;               rule and tracking level >                                    *
;                                                                            *
;  NOTE: While this macro is somewhat unreadable (with lots of cars and      *
;        cdrs), the gained efficiency is worthwhile since this is one of     *
;        most accessed routines in the program.                              *
;                                                                            *
;*****************************************************************************

(defmacro create-edge (args)
;#############################################################################
 '(progn
   (let ((new-edge (newsym 'edge)))

    (setplist new-edge
         (list 'left-vertex  (car ,args)
               'right-vertex (cadr ,args)
               'category     (caddr ,args)
               'needed       (cadddr ,args)
               'contents     (caddddr ,args)
               'rule         (cadddddr ,args)
         )
    )

    new-edge                                            ; return new edge value
   )
  )
)
```

```
;////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                            |
;  MODULE:  UNIFICATION + BINDING                                            |
;  PURPOSE: To provide support routines for feature manipulation.            |
;                                                                            |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////

;****************************************************************************
;                                                                            *
;  FUNCTION: my-unify                                                        *
;  PURPOSE:  To attempt to unify two expressions.                            *
;            If the two expressions are identical then they unify.           *
;            Otherwise, return a set of variable bindings that make the      *
;            two expressions identical when the values for the variables     *
;            are substituted into both expressions.                          *
;  INPUT:    two expressions                                                 *
;  OUTPUT:   t if equivalent, list of variable bindings that make them       *
;            equivalent, or nil if they are not and cannot be made equal.    *
;                                                                            *
;  NB: Unification presupposes that no occurances of typographically         *
;      equal variables are shared between expressions.                       *
;                                                                            *
;****************************************************************************

(define (my-unify a b)
 (cond
  ((or (atom a) (atom b))                          ; not a proposition?
   (let* ((temp1 (cond ((atom a) a) (t b)))        ; exchange if need be
          (temp  (cond ((atom a) b) (t a)))        ;  so a is atomic
          (a temp1)
          (b temp))
    (cond
     ((eq a b) (list '(> k)))                      ; a=b -> ((> k))
     ((variable? a)                                ; is a variable?
      (cond                                        ;  yes
       ((and (listp b) (member a b)) nil)          ;   a is in list b -> fail
       (t (list (list a b)))                       ;   else -> ((a b))
      )                                            ;     "a is bound to b"
     )
     ((variable? b)                                ; is b variable? ->
      (list (list b a))                            ;   ((b a))
     )                                             ;     "b is bound to a"
     (t nil)                                       ; else fail
    )
   )
  )
  (t                                               ; otherwise not a proposition
   (let ((head-bindings (my-unify (head a) (head b))))
    (cond
     ((not head-bindings) nil)                     ; head-bindings nil -> file
     (t
      (let* ((new-a-tail (my-bind (tail a) head-bindings))
             (new-b-tail (my-bind (tail b) head-bindings))
             (tail-bindings (my-unify new-a-tail new-b-tail)))
       (cond
        (tail-bindings                             ; tail-bindings ok?
         (my-compose head-bindings tail-bindings))  ;  -> compose
        (t nil)                                    ; else fail
       )
      )
     )
    )
   )
  )
 )
)

;****************************************************************************
;                                                                            *
;  FUNCTION: variable?                                                       *
;  PURPOSE:  To determine whether the given item is a variable.              *
;                                                                            *
;****************************************************************************

(defmacro variable? (item) '(numberp ,item))

;****************************************************************************
;                                                                            *
;  FUNCTION: variable-name and variable-value                                *
;  PURPOSE:  To return the name or value of the variable in the binding-list *
;                                                                            *
;****************************************************************************

(define (variable-name binding-list)  (first binding-list))
```

```
(define (variable-value binding-list) (second binding-list))

;*****************************************************************************
;                                                                            *
;  FUNCTION: bind                                                            *
;  PURPOSE:  To replace any variable in expression by its value as given     *
;            in the bindings list, if there is one.                          *
;  INPUT:    expression and bindings list.                                   *
;  OUTPUT:                                                                   *
;                                                                            *
;*****************************************************************************

(define (my-bind expression bindings-list)
 (cond
  ((null bindings-list) expression)                  ; finished -> expression
  ((variable-value (head bindings-list))             ; if first bind has a value
   (my-bind                                          ; recurse on tail
     (replace-occurances                             ;  replace all occurances
       expression                                    ;   of first variable in
       (head bindings-list))                         ;   bindings list with value
     (tail bindings-list)                            ; rest of bindings
   )
  )
  (t (my-bind expression (tail bindings-list)))      ; if no varible value
 )                                                   ; go to next binding
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: replace-occurances                                              *
;  PURPOSE:  To replace the occurances of the variable (which is given in    *
;            the binding-list with its value) with the value of the          *
;            variable, if it has a value.                                    *
;                                                                            *
;*****************************************************************************

(define (replace-occurances expression binding-list)
 (cond
  ((null expression) nil)                            ; examined whole expression?
  ((eq (head expression)                             ;  next item is same as
       (variable-name binding-list))                 ;   variable name?
   (cons                                             ;  then replace it
     (variable-value binding-list)                   ;   with its value
     (replace-occurances                             ;  and replace on tail
       (tail expression)
       binding-list)
   )
  )
  (t (cons                                           ; else
       (head expression)                             ;  save the next item as is
       (replace-occurances                           ;  replace on tail
          (tail expression)
          binding-list))
  )
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: my-compose                                                      *
;  PURPOSE:  To replace any variables on the rhs of an = in the first        *
;            binding list with their values from the second binding list.    *
;  INPUT:                                                                    *
;  OUTPUT:   The result of appending the above with a list of any            *
;            bindings in the second binding list involving variables not     *
;            occuring in the first binding list.                             *
;                                                                            *
;*****************************************************************************

(define (my-compose binding-list1 binding-list2)
 (append
   (my-compose2 binding-list1 binding-list2)
   (variables-in2-notin1                             ;   bindings in list2
     (variable-list binding-list1)                   ;   which have variables
     binding-list2                                   ;   not occuring in list1
   )
 )
)

(define (variable-list binding-list)
 (append (mapcar 'car binding-list)                  ; lhs of rules
;         (mapcar 'is-a-variable                     ;  if variables then
           (mapcar 'caddr binding-list)              ;  rhs also
 )
)
```

```
(define (my-compose2 binding-list1 binding-list2)
 (cond
  ((null binding-list2) binding-list1)                    ; finished? -> result
  ((variable-value (head binding-list2))                  ; variable has a value?
   (my-compose2                                            ;  recurse on tail
     (replace-rhs-occurances                               ;  replace all occurances
       binding-list1                                       ;   of first variable in rhs of
       (head binding-list2))                               ;   bindings list with value
     (tail binding-list2)                                  ; rest of binding-list2
   )
  )
  (t (my-compose2 binding-list1 (tail binding-list2)))) ; else if no variable value
 )                                                        ;  then next binding
)

;*************************************************************************
;                                                                        *
;  FUNCTION: variables-in2-notin1                                        *
;  PURPOSE:  To return the sub-list of bindings in list2 which do not have  *
;            variables occuring in list1.                                *
;                                                                        *
;*************************************************************************

(define (variables-in2-notin1 variables-in-list1 list2 )
 (cond
  ((null list2) nil)
  ((member (variable-name (head list2))                   ; next variable in list2
           variables-in-list1)                            ; a variable in list1?
   (variables-in2-notin1                                  ;  recurse
       variables-in-list1                                 ;   same variables
       (tail list2))                                      ;   forget first
  )
  (t (cons                                                ; else
       (head list2)                                       ;  keep head
       (variables-in2-notin1                              ;  filter tail
          variables-in-list1
          (tail list2)))
  )
 )
)

;*************************************************************************
;                                                                        *
;  FUNCTION: replace-rhs-occurances                                      *
;  PURPOSE:  To replace the occurances of the variable (which is given in   *
;            the parameter "binding" along with its value) on the rhs of   *
;            bindings in the binding list with the value of the variable,  *
;            if it has a value.                                          *
;                                                                        *
;*************************************************************************

(define (replace-rhs-occurances bindings-list binding)
 (cond
 ((null bindings-list) nil)                               ; no more binding-list -> stop
  ((eq (second (head bindings-list))                      ;  next item rhs is same as
       (variable-name binding))                           ;   variable name?
   (cons                                                  ;  then replace it with
    (list                                                 ;   a new binding:
     (first (head bindings-list))                         ;    - same lhs variable
     (variable-value binding))                            ;    - its value
    (replace-rhs-occurances                               ;  and replace-rhs on tail
       (tail bindings-list)
       binding)
   )
  )
  (t (cons                                                ; else
       (head bindings-list)                               ;  save the next item as is
       (replace-rhs-occurances                            ;  replace-rhs on tail
          (tail bindings-list)
          binding))
  )
 )
)
```

```
;////////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                               |
;  MODULE:  FEATURE PARSES                                                      |
;  PURPOSE: To provide support for retrieving parses with features.             |
;                                                                               |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////////

;*******************************************************************************
;                                                                              *
;  FUNCTION: output-trees-features                                             *
;  PURPOSE:  To print out a tree listing of the parsings for feature parse.    *
;                                                                              *
;*******************************************************************************

(define (output-trees-features  successful-edges)
;################################################
 (let ((trees (parse-tree-feature-list successful-edges)))
  (blank 2)
  (mapc 'treeprint trees)
 )
)

;*******************************************************************************
;                                                                              *
;  FUNCTION: output-parses-features                                            *
;  PURPOSE:  To print out a nice listing of the parsings for feature parse.    *
;                                                                              *
;*******************************************************************************

(define (output-parses-features successful-edges)
;################################################
 (let ((trees (parse-tree-feature-list successful-edges)))
  (blank 2)
  (princ (eval '(pp ,trees)))
 )
)

;*******************************************************************************
;                                                                              *
;  FUNCTION: parse-tree-feature-list                                           *
;  PURPOSE:  To translate a list of successful edges with their bindings      *
;            into parse trees.                                                 *
;                                                                              *
;*******************************************************************************

(define (parse-tree-feature-list successful-edges)
;################################################
  (mapcar 'make-tree-features successful-edges)
)

;*******************************************************************************
;                                                                              *
;  FUNCTION: make-tree-features                                                *
;  PURPOSE:  To build a complete tree structure from an edge by collecting    *
;            all bindings for that edge, then calling make-tree on all        *
;            the included edges.                                               *
;                                                                              *
;*******************************************************************************

(define (make-tree-features edge-binding-list)          ; parameter consists of:
 (let ((edge         (first  edge-binding-list))        ;  - edge 1st
       (binding-list (second edge-binding-list)))       ;  - binding 2nd
  (cond
   ((is-a-category (first (category-of edge)))          ; lexical edge?
    (list                                               ; list of
     (instantiate (category-of edge) binding-list)      ;  bindings of edge
     (list (contents-of edge)))                         ;  semantics or word
   )
   (t
    (cons
     (instantiate (category-of edge) binding-list)      ;  bindings of edge
;    (mapcar 'make-tree-features (contents-of edge))    ;  trees of all
     (mapcar 'make-tree-features                        ;  trees of all
       (add-binds-all                                   ;   contained edges
         (contents-of edge) (tail binding-list)))       ;   with current bindings
    )
   )
  )
 )
)

;-------------------------------------------------------------------------------
; FUNCTION: add-binds-all                                                       |
; PURPOSE:  to add the given bindings to all items on the edge-list.            |
```

```
;-------------------------------------------------------------------------------

(define (add-binds-all edge-list bindings)
;-----------------------------------------
 (cond
  ((null edge-list) nil)
  (t
   (cons
    (add-binds (head edge-list) bindings)
    (add-binds-all (tail edge-list) bindings)
   )
  )
 )
)

;-------------------------------------------------------------------------------
; FUNCTION: add-binds                                                          |
; PURPOSE:  to add given bindings to the binding list of edge-binds.           |
;-------------------------------------------------------------------------------

(define (add-binds edge-binds bindings-to-add)
;---------------------------------------------
 (list
  (first edge-binds)                              ; the edge
  (append (second edge-binds) bindings-to-add)  ; bindings + bindings to add
 )
)

;*******************************************************************************
;                                                                              *
;  FUNCTION: find-feature-parses                                               *
;  PURPOSE:  To find all the parses from the chart by finding all the          *
;            complete parses starting with edges from start.                   *
;                                                                              *
;*******************************************************************************

(define (find-feature-parses startsymbol chart)
;##############################################
 (let ((start  (first chart))                 ; start = first vertex in chart
       (finish (car (last chart))))           ; finish = last vertex in chart
  (find-all-feature-parses
     startsymbol
     finish
     (edges-out start)                        ; examine the edges from start
  )
 )
)

;*******************************************************************************
;                                                                              *
;  FUNCTION: find-all-feature-parses                                           *
;  PURPOSE:  To recurse on the list of all edges from the start vertex and     *
;            test to see if they meet the three conditions required to be      *
;            a legal parse:                                                    *
;              1 -- edge out must have start symbol label.                     *
;              2 -- the right vertex of edge must be the finish vertex.        *
;              3 -- there must be no more required constituents.               *
;  OUTPUT:   Return the list of edges and their bindings which represent      *
;            the valid parses in the chart.                                    *
;                                                                              *
;*******************************************************************************

(define (find-all-feature-parses startsymbol finish edges-out-of-start)
;######################################################################*
 (cond
  ((null edges-out-of-start) nil)                  ; finished -> quit
  ((and                                            ; parse good if:
    (eq (first                                     ; 1 first of
         (category-of (head edges-out-of-start)))  ; 1  cat of edge is start
        startsymbol)                               ;    symbol and
    (eq (right-vertex-of (head edges-out-of-start)) ; 2 right vertex is
        finish)                                    ;    last vertex and
    (eq (needed-of (head edges-out-of-start))      ; 3 no needed constituents
        nil))
   (cons
         (list
          (head edges-out-of-start)                ; keep head of list with
          (contents-of (head edges-out-of-start))  ;  its bindings
         )
         (find-all-feature-parses                  ; examine tail
            startsymbol
            finish
            (tail edges-out-of-start))
   )
  )
  (t (find-all-feature-parses                      ;  else examine tail
```

```
        startsymbol
        finish
        (tail edges-out-of-start))
   )
  )
 )

;*****************************************************************************
;                                                                            *
;  FUNCTION: collect-all-bindings                                            *
;  PURPOSE:  To collect all the bindings of an edge which are held           *
;            implicitly in its contents field, and explicitly in its         *
;            category.                                                       *
;                                                                            *
;*****************************************************************************

(define (collect-all-bindings edge)
 (cond
  ((is-a-category (category-of edge)) nil)          ; edge lexical? -> nil
  (t (mapcar                                        ; append
       'collect-edge-binding                        ;  collect-all-bindings
       (contents-of edge))                          ;  for each pair
  )                                                 ;  <contained-edge bindings>
 )                                                  ;  in edge contents
)

(define (collect-edge-binding edge-binding)
 (append                                            ; append
   (collect-all-bindings (first edge-binding))      ;  bindings for edge
   (second edge-binding)                            ;  bindings of edge
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: find-non-variable-value                                         *
;  PURPOSE:  To recover the ultimate "real" value for a feature variable.    *
;                                                                            *
;*****************************************************************************

(define (find-non-variable-value feature-variable binding-list)
 (let ((value (find-value feature-variable binding-list))) ; get feature value
  (cond
   ((null value) feature-variable)                                  ; no value? -> nil
   ((not (variable? value)) value)                  ; not variable? -> value
   (t (find-non-variable-value value binding-list))  ; else recurse
  )
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: find-value                                                      *
;  PURPOSE:  To find the value of the given variable in the binding-list     *
;            or nil if it's not in the list.                                 *
;                                                                            *
;*****************************************************************************

(define (find-value variable binding-list)
 (cond
  ((null binding-list) nil)                         ; empty binding-list? -> nil
  ((eq variable (first (head binding-list)))        ; variable matches?
   (second (head binding-list)))                    ;  -> value
  (t (find-value variable (tail binding-list)))     ; else look at tail
 )
)

;*****************************************************************************
;                                                                            *
;  FUNCTION: instantiate                                                     *
;  PURPOSE:  To replace any variable in category by a non-variable value.    *
;            Instantiate deals with both lists and atoms, allowing           *
;            flexibility when calling.                                       *
;                                                                            *
;*****************************************************************************

(define (instantiate category binding-list)
 (cond
  ((null category) nil)                             ; finished? -> nil
  ((atom category)                                  ; if just atom sent
   (cond
    ((variable? category)                           ;  variable? ->
     (find-non-variable-value category binding-list))  ;  give value
    (t category))                                   ;  else return
  )
  (t                                                ; else
```

```
        (cons                                                   ;  construct a list
         (instantiate (head category) binding-list)             ;   keep head
         (instantiate (tail category) binding-list))            ;   instantiate tail
       )
      )
     )
```

```
;////////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                               |
;  MODULE:  FEATURE_PROCESS                                                     |
;  PURPOSE: To provide support for processing edges with features.              |
;                                                                               |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////////

;*******************************************************************************
;                                                                              *
;  FUNCTION: process-features                                                  *
;  PURPOSE:  To process an edge.  If edge is incomplete, process using         *
;            process1.  If edge is complete, process using process2.           *
;            Process1 and process2 described in detail below.                  *
;                                                                              *
;  NB: This is an updated version of process with features. (12/3/86)          *
;*******************************************************************************

(define (process-features edge track-level)
;##########################################
 (cond
  ((incomplete? edge)                                  ; if edge is incomplete
   (process1-f edge
       (complete-edges (edges-out (right-vertex-of edge)))
       track-level)
  )
  ((complete? edge)                                    ; if edge is complete
   (process2-f edge
       (incomplete-edges (edges-in (left-vertex-of edge)))
       track-level)
  )
 )
)


;*******************************************************************************
;                                                                              *
;  FUNCTION: process1-f                                                        *
;  PURPOSE:                                                                    *
;                                                                              *
;*******************************************************************************

(define (process1-f edge complete-edges track-level)
;###################################################
 (cond
  ((null complete-edges) nil)                         ; finished --> stop
  (t
   (let* ((complete (head complete-edges))
          (needs    (needed-of edge))
          (bindings (my-unify (first needs) (category-of complete))))

    (cond
     (bindings                                        ; unification success?
      (cond
       ((equal bindings '((o k)))                     ; no bindings?
        (my-push                                      ;  add new edge to agenda
          (create-edge
           (list
            (left-vertex-of edge)                     ;  left vertex
            (right-vertex-of complete)                ;  right vertex
            (category-of edge)                        ;  category
            (tail needs)                              ;  needed
            (append1 (contents-of edge)               ;  contents edge +
                     (list complete))                 ;    complete edge
            (rule-of edge)                            ;  rule of proposing edge
            track-level                               ;  tracking information
           )
          )
          *agenda*
        )
       )                                              ; else save bindings
       (t                                             ; and instantiate
        (my-push                                      ;  add new edge to agenda
          (create-edge
           (list
            (left-vertex-of edge)                     ;  left vertex
            (right-vertex-of complete)                ;  right vertex
            (instantiate (category-of edge) bindings) ;  category instantiated
                                                      ;   with bindings
            (instantiate (tail needs) bindings)       ;  needed instantiated
                                                      ;   with bindings
            (append1 (contents-of edge)               ;  contents edge +
                     (list complete                   ;    complete edge
                           bindings))                 ;    w/ inc/comp bindings
            (rule-of edge)                            ;  rule of proposing edge
```

```
            track-level                                    ;  tracking information
           )
          )
          *agenda*
         )
        )
       )
      )
     ) ; end COND
     ) ; end of let
     (process1-f edge (tail complete-edges) track-level) ; process tail
    )
   )
  )

;*************************************************************************
;                                                                        *
;  FUNCTION: process2-f                                                  *
;  PURPOSE:                                                              *
;                                                                        *
;*************************************************************************

(define (process2-f edge incomplete-edges track-level)
 (cond
  ((null incomplete-edges) nil)                          ; finished --> stop
  (t
   (let* ((incomplete (head incomplete-edges))
          (needs (needed-of incomplete))
          (bindings (my-unify (first needs) (category-of edge))))
    (cond
     (bindings                                           ; my-unify successful?
      (cond
       ((equal bindings '((o k)))                        ; no bindings?
        (my-push                                         ;  add new edge to agenda
          (create-edge
           (list
            (left-vertex-of incomplete)                  ;  left vertex
            (right-vertex-of edge)                       ;  right vertex
            (category-of incomplete)                     ;  category
            (tail (needed-of incomplete))                ;  needed
            (append1 (contents-of incomplete)            ;  contents incomplete
                     (list edge))                        ;     and edge
            (rule-of incomplete)                         ;  rule of proposing edge
            track-level                                  ;  tracking information
           )
          )
          *agenda*
         )
        )
        (t                                               ; else save bindings
                                                         ; and instantiate
         (my-push                                         ;  add new edge to agenda
          (create-edge
           (list
            (left-vertex-of incomplete)                  ;  left vertex
            (right-vertex-of edge)                       ;  right vertex
            (instantiate (category-of incomplete) bindings) ;  category
                                                         ;   with bindings
            (instantiate (tail (needed-of incomplete))   ; needed
                         bindings)                       ;   with bindings
            (append1 (contents-of incomplete)            ;  contents incomplete
                     (list edge bindings))               ;       + edge symbol
            (rule-of incomplete)                         ;  rule of proposing edge
            track-level                                  ;  tracking information
           )
          )
          *agenda*
         )
        )
       )
      )
     )
    ) ; end of let
    (process2-f edge (tail incomplete-edges) track-level) ; process tail
   )
  )
 )

(define (process2-f1 edge incomplete-edges track-level)
 (cond
  ((null incomplete-edges) nil)                        ; finished --> stop
  ((eq (first (needed-of (head incomplete-edges)))     ; first needed incomplete
       (category-of edge))                             ;  eq cat of edge?
   (my-push                                            ;  add new edge to agenda
     (create-edge
      (list
       (left-vertex-of (head incomplete-edges))        ;  left vertex
```

```
            (right-vertex-of edge)                          ;  right vertex
            (category-of (head incomplete-edges))           ;  category
            (tail (needed-of (head incomplete-edges)))      ;  needed
            (append1 (contents-of (head incomplete-edges)) ;  contents incomplete
                     edge)                                  ;      + edge symbol
            (rule-of (head incomplete-edges))               ;  rule of proposing edge
            track-level                                     ;  tracking information
           )
          )
          *agenda*)
         (process2-f1 edge (tail incomplete-edges) track-level); process tail
        )
        (t                                                    ; else
         (process2-f1 edge (tail incomplete-edges) track-level); process tail
        )
       )
      )
```

SECTION 11

REALISATION MODULE

```
;////////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                               |
;  MODULE:   REALIZATION                                                        |
;  PURPOSE:  To realize a give syntactic tree structure using morphological     |
;            synthesis.                                                         |
;  OWNER:    copywrite Mark T. Maybury, May, 1987.                              |
;                                                                               |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////////


;-------------------------------------------------------------------------------
; FUNCTION:  realize                                                            |
; PURPOSE:   To linearize a syntax tree to a surface string of words by        |
;            recursing down the tree until it reaches the leaves, at which     |
;            point a constituent is realized.                                   |
;-------------------------------------------------------------------------------

(define (realize syntax-tree)
;----------------------------
 (mapcar 'realize-constituent (linearize syntax-tree))
)

;-------------------------------------------------------------------------------
; FUNCTION:  linearize                                                          |
; PURPOSE:   To linearize a syntax tree to a surface string of words with      |
;            their features by recursing down the tree until reaching the       |
;            leaves, at which  point a root and features in returned.           |
;-------------------------------------------------------------------------------

(define (linearize syntax-tree)
;------------------------------
 (cond
  ((null syntax-tree) nil)                          ; completed realization?
  ((is-a-category (first (head syntax-tree)))       ; if next item terminal
   (list
    (append
     (caar (tail syntax-tree))
     (head syntax-tree))))
  (t
   (apply 'append
    (mapcar 'linearize (tail syntax-tree)))         ; else realize the beginning
  )
 )
)

;-------------------------------------------------------------------------------
; FUNCTION: realize-constituent                                                 |
; PURPOSE:  To call the morphological syntesizer to shape the output           |
;           in accordance with the features of the word.                        |
;-------------------------------------------------------------------------------

(define (realize-constituent word-features)
;------------------------------------------
 (morph-syn
  (head word-features)          ; word
  (list (tail word-features))   ; feature list
 )
)
```

```
;////////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                               |
;  MODULE:  MORPHOLOGICAL SYNTHESIZER                                           |
;  PURPOSE: To synthesize the proper surface structure of a root word           |
;           given the proper features (eg plur, past, 3p, etc).                 |
;  OWNER:   copywrite Mark T. Maybury, May, 1987.                               |
;                                                                               |
;  ND: Assumes dictionary format, entry = <syntax semantics realization>        |
;      described in ~/lisp/dictionary/dictionary_macro.l.                       |
;                                                                               |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////////

;-------------------------------------------------------------------------------
; FUNCTION: morph-syn                                                           |
; PURPOSE:  top level routine                                                   |
;-------------------------------------------------------------------------------

(define (morph-syn root entry)
;-----------------------------
 (cond
  ((nounp entry)                                    ; if noun
   (morph-syn-n root entry))
  ((verb? (word-type  entry))                       ; else if verb
   (morph-syn-v root
    (list (cons 'verb (head entry)) (tail entry))))
  ((adjp entry)                                     ; else if adjective
   (morph-syn-adj root entry))
  ((namep entry)                                    ; else if proper noun?
   (morph-syn-name root))
  (t root)                                          ; else return given
 )
)
;-------------------------------------------------------------------------------
; FUNCTION: look-for-irregulars                                                 |
; PURPOSE:  To look up a given root word with features to determine if it       |
;           has an irregular form before automatic morphological synthesis.     |
;-------------------------------------------------------------------------------

(define (look-for-irregulars root entry)
 (let ((irreg-entries (a-list root)))
  (cond
   ((null irreg-entries))                           ; nil if no irreg-entries
   ((head (matches entry irreg-entries)))           ; check for matches - return
  )                                                 ;   just first realization
 )                                                  ;   UPDATE this to say
)                                                   ;   choose randomly or intelligently

;-------------------------------------------------------------------------------
; FUNCTION: matches                                                             |
; PURPOSE:  Takes list1 and returns the items that match it in list 2.          |
;           List2 is a description of form < syntax semantics realization>      |
;-------------------------------------------------------------------------------

(define (matches list1 list2)
 (cond
  ((null list2))
  ((my-unify list1 (syntax (head list2)))     ; if syntax of first item matches
   (cons
    (realization (head list2))                ; save it and
    (matches list1 (tail list2)))             ; recurse on tail
  )
  (t (matches list1 (tail list2)))            ; else recurse on tail
 )
)

;-------------------------------------------------------------------------------
; FUNCTION: morph-syn-n                                                         |
; PURPOSE:  To synthesize the proper form of a noun given root and features.    |
;-------------------------------------------------------------------------------

(define (morph-syn-n root entry)
;-------------------------------
 (cond
  ((and (eq (noun-type entry)  'count)
        (eq (noun-count entry) 'plur))
   (make-n-plural root))
  (t root)
 )
)
;-------------------------------------------------------------------------------
; FUNCTION: morph-syn-v                                                         |
; PURPOSE:  To synthesize the proper form of a verb given root and features.    |
;-------------------------------------------------------------------------------
```

```
(define (morph-syn-v root entry)
;-------------------------------
 (cond
  ((member (verb-type entry) '(copula aux modal do think-know have-v))
   root)                                  ; return root if irregular
  ((eq (verb-ing entry) 'ing)             ; present participle?
   (make-v-ing root))
  ((and (eq (verb-count entry) 'sing3p)   ; 3rd person singular?
        (eq (verb-tense entry) 'pres))
   (make-v-3ps root))
  ((eq (verb-tense entry) 'past)          ; past tense?
   (make-v-past root))
  (t root)                                ; else return root
 )
)

;-----------------------------------------------------------------------------
; FUNCTION: morph-syn-adj                                                     |
; PURPOSE:  To synthesize the proper form of a adj given root and entry.     |
;-----------------------------------------------------------------------------

(define (morph-syn-adj root entry)
;---------------------------------
 (cond
  ((eq (adj-type entry) 'comparative) (add root 'er))
  ((eq (adj-type entry) 'superlative) (add root 'est))
  (t root)
 )
)
;-----------------------------------------------------------------------------
; FUNCTION: morph-syn-name                                                    |
; PURPOSE:  To synthesize the proper form of a name given root (capitalize) |
;-----------------------------------------------------------------------------

(define (morph-syn-name root entry)
;----------------------------------
 (capitalize root)
)

;-----------------------------------------------------------------------------
;  FUNCTION: add                                                              |
;  PURPOSE:  To add or concatenate a given ending to a word.                  |
;-----------------------------------------------------------------------------

(define (add word ending)
;------------------------
 (cond
  ((null word))                                          ; no word -> don't add
  ((null ending))                                        ; no ending -> nil
  (t (implode (append (explode word) (explode ending))))); otherwise add it on
 )
)

;-----------------------------------------------------------------------------
; FUNCTION: make-n-plural                                                     |
; PURPOSE:  To add the appropriate morphological ending to pluralize a noun.|
;-----------------------------------------------------------------------------

(define (make-n-plural root)
;--------------------------
 (cond
  ((eq (last (explode root)) 's) (add root 'es))
  (t (add root 's))
 )
)

;-----------------------------------------------------------------------------
; FUNCTION: make-v-3ps                                                        |
; PURPOSE:  To add the appropriate morphological ending to make verb 3ps.    |
;-----------------------------------------------------------------------------

(define (make-v-3ps root)  (add root 's))
;------------------------

;-----------------------------------------------------------------------------
; FUNCTION: make-v-ing                                                        |
; PURPOSE:  To add the appropriate morphological ending to make verb past.   |
;-----------------------------------------------------------------------------

(define (make-v-ing root)  (add root 'ing))
;------------------------
;
; lie -> lying,  fly -> flying,  take -> taking

;-----------------------------------------------------------------------------
; FUNCTION: make-v-past                                                       |
; PURPOSE:  To add the appropriate morphological ending to make verb past.   |
```

```
;-------------------------------------------------------------------------------

(define (make-v-past root)  (add root 'ed))
;--------------------------
```

```
;////////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                               |
;  MODULE:   SURFACE FORM                                                       |
;  PURPOSE:  To present syntactic generator output in a clean surface form.     |
;  OWNER:    copywrite Mark T. Maybury, June, 1987.                             |
;                                                                               |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////////


;-------------------------------------------------------------------------------
; FUNCTION: surface-form                                                        |
; PURPOSE:  to produce a sentence with proper othographical presentation        |
;           following punctuation rules and spacing conventions.                |
;-------------------------------------------------------------------------------

(define (surface-form lex-punct-list)
;------------------------------------
 (output-surface-form
  (append
   (list (capitalise (first lex-punct-list)))
   (tail lex-punct-list)
   '(period)
  )
 )
)

;-------------------------------------------------------------------------------
; FUNCTION: output-surface-form                                                 |
; PURPOSE:  to output a sentence with proper othographical presentation         |
;           following punctuation rules and spacing conventions.                |
;-------------------------------------------------------------------------------

(define (output-surface-form lex-punct-list)
;-------------------------------------------
 (cond
  ((null lex-punct-list) (msg N))                    ; move to new line
  ((punctuation? (second lex-punct-list))            ; next item punctuation
   (princ (first lex-punct-list))                    ; output next word
   (print-punct (second lex-punct-list))             ; output punctuation
   (princ " ")                                       ; output space
   (output-surface-form
    (tail (tail lex-punct-list))) ; recurse on rest
  )
  (t
   (princ (first lex-punct-list))                    ; output next word
   (princ " ")                                       ; output space
   (output-surface-form (tail lex-punct-list))       ; recurse on rest
  )
 )
)

;-------------------------------------------------------------------------------
; FUNCTION: capitalize                                                          |
; PURPOSE:  to convert the first letter of the given word into a capital.       |
;-------------------------------------------------------------------------------

(define (capitalize word)
;------------------------
 (implode (cons (capital word) (tail (explode word))))
)


;-------------------------------------------------------------------------------
; FUNCTION: capital                                                             |
; PURPOSE:  to return the first letter of word capitalized (if not already).    |
;-------------------------------------------------------------------------------

(define (capital word)
;---------------------
 (cond
  ((<= (car (substringn word 1)) 90)           ; capital letter? 65 <= code <= 90
   (car (explode word)))                       ;  return it unchanged
  (t (ascii (- (car (substringn word 1)) 32))) ; else capitalize first letter
 )
)

;-------------------------------------------------------------------------------
; FUNCTION: print-punct                                                         |
; PURPOSE:  to print the proper punctuation given a keyword.                    |
;-------------------------------------------------------------------------------

(define (punctuation? item)
;--------------------------
 (member item '(comma period colon question-mark exclamation-point)))
```

```
(define (print-punct punct)
;--------------------------
 (cond
  ((eq punct 'comma) (princ '\,))
  ((eq punct 'period) (princ '\.))
  ((eq punct 'colon) (princ ':))
  ((eq punct 'question-mark) (princ '?))
  ((eq punct 'exclamation-point) (princ '!))
 )
)
```

SECTION 12

KNOWLEDGE SOURCES

SECTION 12.1

MONTAGUE GRAMMAR

```
;=============================================================================
;                                                                            |
;                          MONTAGUE SEMANTIC GRAMMAR                         |
;                                                                            |
; Phrase structures are grouped by classification.  Each grammar rule is     |
; followed by a English example in the Neuropsychology domain.               |
;                                                                            |
; GRAMMAR SYNTAX                                                             |
; --------------                                                             |
; grammar -> <rule>*                                                         |
;  rule -> <rule name> <syntax> <semantics>                                  |
;   rule name -> <string of characters representing rule>                    |
;   syntax    -> <category>*  where head is lhs rule and tail is rhs rule    |
;                             head must be a non-terminal                    |
;    category -> <grammar-symbol> <feature>*                                 |
;     grammar-symbol -> eg s, vp, np, noun, empty-symbol                     |
;     feature        -> <feature-constant> | <feature-variable>              |
;   semantics -> <lambda-calculus-first-order-logical-form>                  |
;                                                                            |
;=============================================================================

(setq *grammar* '(

;=============================================================================
;  DECLARATIVE STATEMENTS, WH AND YES/NO QUESTIONS                           |
;=============================================================================

;-----------------------------------------------------------------------------
; Declarative Sentences                                                      |
;-----------------------------------------------------------------------------

( s<dec>->np+vp      ([s (type declarative) (voice active)]
                       [np (count 1) (person 2) (gender 4)]
                       [vp (count 1) (person 2) (tense 3) (voice active)]]
                      (some (_e) (np (vp _e))) )

  ; "The patient reads slowly"
  ; --------------------------
  ; "The doctor investigates the patient's left hemisphere."
  ; ---------------------------------------------------------
  ; NB:  There are theoretical problems with the event variable.
  ;      Unfortunately, not all verbs are single events and can refer
  ;      to continous or repeated action. eg "The man rented the car for a week."

( s<dec>->np+vp<pas> ([s (type declarative) (voice passive)]
                       [np (count 1) (person 2) (gender 4)]
                       [vp (count 1) (person 2) (tense 3) (voice passive)]]
                      (some (_e) (np (vp _e))) )

  ; "the hemisphere is contained in the brain"
  ; -------------------------------------------

;-----------------------------------------------------------------------------
; Connected Sentences                                                        |
;-----------------------------------------------------------------------------

( s->s+conn+s        ([s (type 1) (voice 2)]
                       [s (type 1) (voice 2)]
                       [connective]
                       [s (type 3) (voice 4)]]          (some (_e) (and s s)) )

  ; "The family is concerned because the patient has Alzheimer's disease."
  ; -----------------------------------------------------------------------

;=============================================================================
;  SIMPLE NOUN PHRASES                                                       |
;=============================================================================

( np->det+n1         ([np (count 1) (person p3) (gender 7)]
                       [determiner (type 2) (count 1) (kind 3) (extension 4)
                                   (negative 5) (number 6)]
                       [n1 (type 1) (gender 7)])
                      (determiner n1))

  ; "these men" "a doctor" "every patient"
  ; --------------------------------------

( np->num+n1         ([np (count 1) (person p3) (gender 2)]
                       [number (type 1)] [n1 (type 1) (gender 2)])
                      (number n1))

( np->proper-noun    ([np (count 1) (person p3) (gender 2)]
                       [proper-noun (count 1) (gender 2)]]
                      (L (_P) (_P proper-noun)) )
  ; "Michelle"
  ; -----------
```

```
( np->mass-noun      ([np (count sing3p) (person p3) (gender 2)]
                       [noun (count mass) (person sing3p) (gender 2)])
                     (L (_P) (_P noun)) )
  ; "patients"
  ;~~~~~~~~~~

( np->noun+mass      ([np (count sing3p) (person p3) (gender 2)]
                       [noun (count 4) (type 5) (gender 6)]
                       [noun (count mass) (person sing3p) (gender 2)])
                     (L (_P) (_P noun)) )

( np->pronoun        ([np (count 1) (person 2) (gender 3)]
                       [pronoun (type pers) (count 1) (kind subj) (person 2) (gender 3)])
                     (L (_P) (_P pronoun)) )
  ; "I" "you" "she"
  ;~~~~~~~~~~~~~~~

( np->np+pp           ([np (count 1) (person 2) (gender 3)]
                        [np (count 1) (person 2) (gender 3)] [pp])
                       (L (_x) (and (n1 _x) (pp _x))) )

  ; "the result of the exam"
  ;~~~~~~~~~~~~~~~~~~~~~~~

( np->np:np          ([np (count 1) (person 2) (gender 3)]
                       [np (count 1) (person 2) (gender 3)]
                       [colon]
                       [np (count 4) (person 5) (gender 6)])
                     (L (_P) (_P np)) )

  ; "three parts:  the hammer, anvil, and stirrup"
  ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

( np->np-comma-np    ([np (count plur) (person 2) (gender 3)]
                       [np (count 1) (person 2) (gender 3)]
                       [comma]
                       [np (count 4) (person 5) (gender 6)])
                     (L (_P) (_P np)) )

  ; "the hammer, the anvil"
  ;~~~~~~~~~~~~~~~~~~~~~~~

( np->np-conj-np     ([np (count plur) (person 2) (gender 3)]  ; NB: count plur
                       [np (count 1) (person 2) (gender 3)]
                       [conjunction (type 4)]
                       [np (count 5) (person 6) (gender 7)])
                     (L (_P) (_P np)) )
                      ; gen "a, b and c" or "b and c" but not "a, b, and c"

  ; "the anvil and the stirrup"
  ;~~~~~~~~~~~~~~~~~~~~~~~~~~

( np->np+rel         ([np (count 1) (person p3) (gender 2)]
                       [np (count 1) (person p3) (gender 2)]
                       [comma]
                       [rel (type 3)]    ; which, therefore, for example
                       [comma])                                    (np rel))

  ; "the patient who injested the poison"
  ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

; Not sure if this is a theoretical linguistic insight
; or a hack below but it is efficient:

( rel->connective    ([rel (type connective)] [connective])      (connective))

  ; "which"
  ;~~~~~~~

( rel->connective2   ([rel (type 1)]
                       [connective (type 1)]
                       [connective (type 1)])                    (connective))
  ; "for example"
  ;~~~~~~~~~~~~~

;=============================================================================
;  NOUN GROUPS                                                              |
;=============================================================================

( n1->noun           ([n1 (type 1) (gender 3)]
                       [noun (count 2) (type 1) (gender 3)])           (noun))
  ; "alcohol"
  ;~~~~~~~~~

( n1->noun+noun      ([n1 (type 1) (gender 3)]
```

```
                              [noun (count 4) (type 5) (gender 6)]
                              [noun (count 2) (type 1) (gender 3)])          (noun))

  ; "hemisphere region"
  ;~~~~~~~~~~~~~~~~~~~~

( n1->adjp<attr>+n1 ([n1 (type 1) (gender 2)]
                     [adjp (type attributive)] [n1 (type 1) (gender 2)])
                    (L (_x) (and (adjp _x) (n1 _x))) )

  ; "slow recognition"
  ;~~~~~~~~~~~~~~~~~~~

( n1->adjp<sup>+n1  ([n1 (type 1) (gender 2)]
                     [adjp (type superlative)] [n1 (type 1) (gender 2)])
                    (L (_x) (and (adjp _x) (n1 _x))) )

  ; "bigger feet"
  ;~~~~~~~~~~~~~~

;=============================================================================
;  PREPOSITIONAL PHRASES                                                      |
;=============================================================================

( pp->prep+np       ([pp]
                     [preposition] [np (count 1) (person 2) (gender 3)])
                    (preposition np))

  ; "for comprehension"
  ;~~~~~~~~~~~~~~~~~~~~

( pp->prep+np       ([pp]
                     [preposition (type 1)] [preposition (type 1)]
                     [np (count 2) (person 3) (gender 4)])
                    (preposition np))

  ; "located in the left occipital lobe"
  ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

( pp->prep+number   ([pp]
                     [preposition] [number (type 1)])
                    (preposition np))

  ; (a test value) "of one"
  ;~~~~~~~~~~~~~~~~~~~~~~~

;=============================================================================
;  ADJECTIVE PHRASES                                                          |
;=============================================================================

( adjp->adjective   ([adjp (type 1)]
                     [adjective (type 1)])
                    adjective)
  ; "ill"
  ;~~~~~~

( adjp->adj+adjp    ([adjp (type 1)]
                     [adjective (type 1)] [adjp (type 1)])
                    (L (_x) (and (adjective _x) (adjp _x))) )
  ; "right parietal"
  ;~~~~~~~~~~~~~~~~~

;=============================================================================
;  VERB PHRASES                                                               |
;=============================================================================

;-----------------------------------------------------------------------------
;  to have                                                                    |
;-----------------------------------------------------------------------------

( vp->have+np       ([vp (count 1) (person 2) (tense 3) (voice active)]
                     [have-v (count 1) (tense 3) (person 4)]
                     [np (count 5) (person 6) (gender 7)])       (have-v np))

  ; "has the gravest condition"
  ;~~~~~~~~~~~~~~~~~~~~~~~~~~~

;-----------------------------------------------------------------------------
;  Adverbial Phrases and Regular Intransitive/Transitive Verbs                |
;-----------------------------------------------------------------------------

( vp->trans+np      ([vp (count 1) (person 2) (tense 3) (voice active)]
                     [trans (count 1) (tense 3) (person 4)]
                     [np (count 5) (person 6) (gender 7)])        (trans np))
```

```
          ; "contains three regions"
          ;-------------------------

        ( vp->vp+pp     ([vp (count 1) (person 2) (tense 3) (voice 4)]
                          [vp (count 1) (person 2) (tense 3) (voice 4)]
                          [pp])
                        (L (_e) (L (_x) (and ((vp _e) _x) (pp _e)))) )

          ; "has a pain in the head"
          ;-------------------------

        ( vp->intrans   ([vp (count 1) (person 2) (tense 3) (voice active)]
                         [intrans (count 1) (tense 3) (person 4)])
                        intrans)
          ; "snores"
          ;---------

        ( vp->adv+vp    ([vp (count 1) (person 2) (tense 3) (voice 4)]
                         [adverb]
                         [vp (count 1) (person 2) (tense 3) (voice 4)])
                        (L (_e) (L (_x) (and ((vp _e) _x) (adverb _e)))) )
          ; "quickly eats"
          ;---------------

        ( vp->vp+adv    ([vp (count 1) (person 2) (tense 3) (voice 4)]
                         [vp (count 1) (person 2) (tense 3) (voice 4)]
                         [adverb])
                        (L (_e) (L (_x) (and ((vp _e) _x) (adverb _e)))) )
          ; "eats fast"
          ;------------

        ( vp->trans+prep+np ([vp (count 1) (person 2) (tense 3) (voice active)]
                             [trans (count 1) (tense 3) (person 4) (prep 5)]
                             [preposition (type 5)]
                             [np (count 6) (person p3) (gender 2)])
                            (trans (preposition np)) )

          ; "bumps into the wall"
          ;----------------------

        ;-------------------------------------------------------------------------------
        ;  to be                                                                        |
        ;-------------------------------------------------------------------------------

         ; drop event variable from copula clause

        ( vp->copula+adjp<attr>  ([vp (count 1) (person 3) (tense 2) (voice active)]
                                  [copula (count 1) (tense 2) (person 3)]
                                  [adjp (type attributive)])
                                 (L (_e) (L (_x) (adjp _x))) )

          ; "are local" "is visual"
          ;------------------------
          ; treat adj verbing as predicate and plunk in front of the variable _x

        ( vp->copula+verbing     ([vp (count 1) (person 3) (tense 2) (voice active)]
                                  [copula (count 1) (tense 2) (person 3)]
                                  [intrans (count 4) (tense 5) (person 6) (participle ing)])
                                 (L (_e) (L (_x) (intrans _x))) )
          ; "is dying"
          ;------------
          ; incomplete linguistic treatment

        ( vp->copula+np      ([vp (count 1) (person 3) (tense 2) (voice active)]
                               [copula (count 1) (tense 2) (person 3)]
                               [np (count 4) (person 5) (gender 6)])   (L (_e) (copula np)) )

          ; "am the best doctor"
          ;---------------------

        ( vp->copula+pp      ([vp (count 1) (person 3) (tense 2) (voice active)]
                               [copula (count 1) (tense 2) (person 3)]
                               [pp])                                  (L (_e) (L (_x) (pp _e))) )
          ; "is from the alcohol"
          ;----------------------

        ( vp<pas>->copula+v+np([vp (count 1) (person 3) (tense 2) (voice passive)]
                               [copula (count 1) (tense 2) (person 3)]
                               [trans (count 1) (tense 4) (person 5) (form en)]
                               [preposition (type en)]
                               [np (count 6) (person 7) (gender 8)])       (L (_e) (copula np)) )

          ; "is indicated by the comprehension test"
          ;-----------------------------------------

         )
```

SECTION 12.2

NEUROPSYCHOLOGY DICTIONARY AND KNOWLEDGE BASE

```
;////////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                               |
;  MODULE:   KERNAL LEXICON                                                     |
;  PURPOSE:  To serve as a base dictionary from which to build a domain         |
;            lexicon.  The ideas is to develop a tool which will accelerate     |
;            to man-power intensive task of lexicon generation.                 |
;                                                                               |
;  OWNER:    copywrite Mark T. Maybury, July, 1987.                             |
;                                                                               |
;  FORMAT:   < token syntax sematics realization >                              |
;                                                                               |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////////

(erase-dictionary)                              ; defined in makedictionary
(mapc 'make-dictionary-entry '(

;-------------------------------------------------------------------------------
;                               NUMBERS                                         |
;-------------------------------------------------------------------------------

 (one   (number sing3p) (lexical representation of number 1) one)
 (two   (number plur) (lexical representation of number 2) two)
 (three (number plur) (lexical representation of number 3) three)
 (four  (number plur) (lexical representation of number 4) four)
 (five  (number plur) (lexical representation of number 5) five)
 (six   (number plur) (lexical representation of number 6) six)
 (seven (number plur) (lexical representation of number 7) seven)
 (eight (number plur) (lexical representation of number 8) eight)
 (nine  (number plur) (lexical representation of number 9) nine)
 (ten   (number plur) (lexical representation of number 10) ten)

;-------------------------------------------------------------------------------
;                           PROPER NOUNS                                        |
;-------------------------------------------------------------------------------

 (mark (proper-noun sing3p masculine) me mark)
 (michelle (proper-noun sing3p feminine) her michelle)

;-------------------------------------------------------------------------------
;                               VERBS                                           |
;-------------------------------------------------------------------------------

    ; ** the verb "to be"

 (be    (verb copula sing   pres p1)
        (L (_P) (L (_WH) (_P (L (_y) (equal _WH _y))))) am)
 (be    (verb copula sing3p pres p3)
        (L (_P) (L (_WH) (_P (L (_y) (equal _WH _y))))) is)
 (be    (verb copula plur pres p3)
        (L (_P) (L (_WH) (_P (L (_y) (equal _WH _y))))) are)

 (have (verb have-v sing pres p1) (to own or posess) have)
 (have (verb have-v plur   pres p1) (to own or posess) have)
 (have  (verb have-v sing3p pres p3) (to own or posess - irregular 3p sing) has)
; for interpretation:
; (has  (verb have-v sing3p pres p3) (to own or posess - irregular 3p sing) has)

; (contain (verb trans 1 pres 2) (restricted or otherwise limited) contain)
 (contain (verb trans plur   pres p3) (restricted or otherwise limited) contain)
 (contain (verb trans sing3p pres p3) (restricted or otherwise limited) contain)

; (indicate (verb trans 1 pres 2) (telling) indicate)
 (indicate (verb trans sing3p pres p1) (telling) indicate)
 (indicate (verb trans plur   pres p3) (telling) indicate)

; (function (verb trans 1 pres 2) (telling) function)
 (function (verb trans sing3p pres p1) (telling) function)
 (function (verb trans plur   pres p3) (telling) function)

; (suffer (verb trans 1 pres from)  ; removed plur
;         (L (_np) (L (_e) (L (_WH) (_np (L (_y) (suffer _WH _y _e)))))) suffer)


;-------------------------------------------------------------------------------
;                               NOUNS                                           |
;-------------------------------------------------------------------------------


 ; Domain Specific Taxonomy
 ; Fault Classification/System Components

 ; Capabilities/Symptoms
```

```
; Fault Diagnosis Lexical Entries

(example (noun mass 1 neuter) example example)

(function (noun count 1 neuter) function function)
(disorder (noun count 1 neuter) disorder disorder)
(decision (noun count 1 neuter) decision decision)
(diagnosis (noun count 1 neuter) diagnosis decision)
(human    (noun count 1 neuter) human human)

; Entity/Relationship and Frame Knowledge Representation Lexis

(entity      (noun count 1 neuter) entity entity)
(sub-class   (noun count 1 neuter) sub-class sub-class)
(value       (noun count 1 neuter) value value)
(damage      (noun mass 1 neuter) damage damage)
(importance (noun count 1 neuter) importance importance)
(symptom       (noun count 1 neuter) symptom symptom)
(test        (noun count 1 neuter) test test)
(observation (noun count 1 neuter) observation observation)

;-------------------------------------------------------------------------
;                              ADJECTIVES                                 |
;-------------------------------------------------------------------------

(similar (adjective attributive) similar similar)
(different (adjective attributive) different different)
(relative (adjective attributive) relative relative)

(slow (adjective attributive) slow slow)
(fast (adjective attributive) fast fast)

;-------------------------------------------------------------------------
;                              DETERMINERS                                |
;-------------------------------------------------------------------------
; ** articles
(a      (determiner count sing3p indefart notof noneg nonum) (article before consonant) a)
(an     (determiner count sing3p indefart notof noneg nonum) (article before vowel) an)
(the    (determiner count 1 defart notof noneg nonum) (sing/plur form of the) the)

;-------------------------------------------------------------------------
;                              PREPOSITIONS                               |
;-------------------------------------------------------------------------

(to (preposition) (toward or in the direction of) to)
(in (preposition) (inner or inward location) in)
(with (preposition) (connection or association) with)
(from (preposition) (place of origin) from)
(of (preposition) (place of origin) of)
(for (preposition) (indicating purpose) for)

(located (preposition located-in) (located-in) located)
(in (preposition located-in) (located-in) in)

;-------------------------------------------------------------------------
;                              PRONOUNS                                   |
;-------------------------------------------------------------------------

(he   (pronoun pers sing3p subj p3 masculine) (male) he)
(she  (pronoun pers sing3p subj p3 feminine) (female) she)
(it   (pronoun pers sing3p subj p3 neuter) (a thing) it)
; (they (pronoun pers plur   subj p3 neuter) (a group of others) they)

; (him  (pronoun pers sing3p obj p3) (a male viewed objectively) him)
; (her  (pronoun pers sing3p obj p3) (a female viewed objectively) her)
; (it   (pronoun pers sing3p obj p3) (a thing viewed objectively) it)
; (them (pronoun pers plur   obj p3) (a group of others) them)

; (his  (pronoun poss sing3p obj p3) (belonging to a male viewed objectively) his)
; (her  (pronoun poss sing3p obj p3) (belonging to a female viewed objectively) her)
; (its  (pronoun poss sing3p obj p3) (belonging to a thing viewed objectively) its)
; (their (pronoun poss plur  obj p3) (belonging to a group of others) their)

; (his  (pronoun poss sing3p subj p3) (belonging to a male viewed subjectively) his)
; (hers (pronoun poss sing3p subj p3) (belonging to a female viewed subjectively) hers)
; (its  (pronoun poss sing3p subj p3) (belonging to a thing viewed subjectively) its)
; (theirs (pronoun poss plur subj p3) (belonging to a group of others) theirs)

; ** relative pronouns
; (that (pronoun rel) (The ball that is red) that)
; (who (pronoun rel) (The patient who died) who)
; (which (pronoun rel) (The book which burned) which)

; ** demonstrative pronouns
; (this  (pronoun demonstr sing3p) (this book) this)
; (that  (pronoun demonstr sing3p) (that book) that)
```

```
; (these (pronoun demonstr plur) (these books) these)
; (those (pronoun demonstr plur) (those books) those)


;-------------------------------------------------------------------------
;                              PUNCTUATION                                |
;-------------------------------------------------------------------------

 (comma  (comma) comma comma)
 (period (period) period period)
 (colon  (colon) colon colon)
;(exclamation-point  (exclamation-point) exclamation-point exclamation-point)
;(question-mark  (question-mark) question-mark question-mark)

;-------------------------------------------------------------------------
;                              CONJUNCTIONS                               |
;-------------------------------------------------------------------------

 (and (conjunction coord) (intersection) and)
 (or (conjunction coord) (union) or)
 (but (conjunction coord) (qualification) but)
; (before (conjunction subord) (pre-temporal) before)
; (after (conjunction subord) (post-temporal) after)
; (because (conjunction subord) (causality) because)

;                              CONNECTIVES
; don't know where to put this:
;; (there (pronoun pers plur  subj p3 neuter) (there are) there)
 (for    (connective for-example) for for)
 (example  (connective for-example) example example)
 (instance (connective) instance instance)
 (therefore (connective) therefore therefore)
 (because (connective) because because)
))
```

```
;////////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                              |
;  MODULE:   NEUROPSYCHOLOGICAL LEXICON                                        |
;  PURPOSE:  To erase any current dictionary and load in a lexicon of          |
;            neurophysiology and neuropsychology.                              |
;                                                                              |
;  OWNER:    copywrite Mark T. Maybury, July, 1987.                            |
;                                                                              |
;  FORMAT:   < token syntax sematics realization >                             |
;                                                                              |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////////

(erase-dictionary)                              ; defined in makedictionary
(mapc 'make-dictionary-entry '(

;-------------------------------------------------------------------------------
;                              NUMBERS                                         |
;-------------------------------------------------------------------------------

 (one   (number sing3p) (lexical representation of number 1) one)
 (two   (number plur) (lexical representation of number 2) two)
 (three (number plur) (lexical representation of number 3) three)
 (four  (number plur) (lexical representation of number 4) four)
 (five  (number plur) (lexical representation of number 5) five)
 (six   (number plur) (lexical representation of number 6) six)
 (seven (number plur) (lexical representation of number 7) seven)
 (eight (number plur) (lexical representation of number 8) eight)
 (nine  (number plur) (lexical representation of number 9) nine)
 (ten   (number plur) (lexical representation of number 10) ten)

;-------------------------------------------------------------------------------
;                           PROPER NOUNS                                       |
;-------------------------------------------------------------------------------
 (mark (proper-noun sing3p masculine) me mark)
 (michelle (proper-noun sing3p feminine) her michelle)
 (korsakoffs (proper-noun sing3p neuter) korsakoffs korsakoffs)
 (huntingtons (proper-noun sing3p neuter) huntingtons huntingtons)
 (alzheimers (proper-noun sing3p neuter) alzheimers alzheimers)

;-------------------------------------------------------------------------------
;                              VERBS                                           |
;-------------------------------------------------------------------------------

    ; ** the verb "to be"

 (be    (verb copula sing   pres p1)
        (L (_P) (L (_WH) (_P (L (_y) (equal _WH _y))))) am)
 (be    (verb copula sing3p pres p3)
        (L (_P) (L (_WH) (_P (L (_y) (equal _WH _y))))) is)
 (be    (verb copula plur pres p3)
        (L (_P) (L (_WH) (_P (L (_y) (equal _WH _y))))) are)

 (have (verb have-v sing pres p1) (to own or posess) have)
 (have (verb have-v plur   pres p1) (to own or posess) have)
 (have  (verb have-v sing3p pres p3) (to own or posess - irregular 3p sing) has)

 (contain (verb trans plur   pres p3) (restricted or otherwise limited) contain)
 (contain (verb trans sing3p pres p3) (restricted or otherwise limited) contain)

 (indicate (verb trans plur   pres p3) (telling) indicate)
 (indicate (verb trans sing3p pres p1) (telling) indicate)

 (function (verb trans plur   pres p3) (telling) function)
 (function (verb trans sing3p pres p1) (telling) function)

 (manifest (verb trans sing3p pres p1) (evident) manifest)
 (made (verb trans sing3p past p1) (creating or performing) made)

 (suffer (verb trans 1 pres from)
         (L (_np) (L (_e) (L (_WH) (_np (L (_y) (suffer _WH _y _e))))))) suffer)

;-------------------------------------------------------------------------------
;                              NOUNS                                           |
;-------------------------------------------------------------------------------


 ; Neurophysiology

 (organ (noun count sing3p neuter) (cell-based functioning sub component) organ)
 (brain (noun count sing3p neuter) region brain)
 (hemisphere (noun count sing3p neuter) region hemisphere)
 (left-hemisphere (noun count sing3p neuter) region  left-hemisphere)
 (right-hemisphere (noun count sing3p neuter) region right-hemisphere)
 (lfrontal (noun count sing3p  neuter) lobe left-frontal)
```

```
(rfrontal (noun count sing3p neuter) lobe right-frontal)
(lparietal (noun count sing3p neuter) lobe left-parietal)
(rparietal (noun count sing3p neuter) lobe right-parietal)
(lsubcortex (noun count sing3p neuter) lobe left-subcortex)
(rsubcortex (noun count sing3p neuter) lobe right-subcortex)
(ltemporal (noun count sing3p neuter) lobe left-temporal)
(rtemporal (noun count sing3p neuter) lobe right-temporal)
(loccipital (noun count sing3p neuter) lobe left-occipital)
(roccipital (noun count sing3p neuter) lobe right-occipital)
(skull (noun count 1 neuter) (cranial container and protector) skull)
(body       (noun count 1 neuter) body body)
(region     (noun count 1 neuter) region region)
(lobe       (noun count 1 neuter) lobe lobe)

; capabilities/symptoms

(motor-response (noun mass 1 neuter) motor-response motor-response)
(vision (noun mass 1 neuter) vision vision)
(sensation (noun mass 1 neuter) sensation sensation)
(stability (noun mass 1 neuter) stability stability)
(stability-detection (noun mass 1 neuter) stability-detection stability-detection)
(control (noun mass 1 neuter) control control)
(memory (noun mass 1 neuter) memory memory)
(naming (noun mass 1 neuter) naming naming)
(instability (noun mass 1 neuter) instability instability)
(personality (noun mass 1 neuter) personality personality)
(sex-activity (noun mass 1 neuter) sexual-activity sexual-activity)
(feature-recognition (noun mass 1 neuter) function feature-recognition)
(gestalt-understanding (noun mass 1 neuter) function gestalt-understanding)
(intrapersonal-behavior (noun mass 1 neuter) intrapersonal-behavior intrapersonal-behavior)
(understanding (noun mass 1 neuter) consciousness understanding)
(language (noun mass 1 neuter) language language)
(comprehension (noun mass 1 neuter) comprehension comprehension)
(mental-control (noun mass 1 neuter) mental-control mental-control)
(immediate-recall (noun mass 1 neuter) immediate-recall immediate-recall)
(wisconsin (noun mass 1 neuter) wisconsin Wisconsin)
(l-cog-flexibility (noun mass 1 neuter) l-cog-flexibility left-cognitive-flexibility)
(r-cog-flexibility (noun mass 1 neuter) r-cog-flexibility right-cognitive-flexibility)
(m-n (noun mass 1 neuter) m-n m-n)
(m-n-perseveration (noun mass 1 neuter) m-n-perseveration m-n-perseveration)
(loops (noun mass 1 neuter) loops loop)
(loops-perseveration (noun mass 1 neuter) loops-perseveration loops-perseveration)
(stm (noun mass 1 neuter) stm short-term-memory)
(drawings (noun mass 1 neuter) drawings drawings)
(scribbles (noun mass 1 neuter) scribbles scribbles)
(pencil (noun count 1 neuter) pencil pencil)
(construction (noun mass 1 neuter) construction construction)
(example (noun mass 1 neuter) example example)

(history (noun count 1 neuter) history history)
(function (noun count 1 neuter) function function)
(location (noun count 1 neuter) location location)
(instrument (noun count 1 neuter) instrument instrument)
(disorder (noun count 1 neuter) disorder disorder)
(failure (noun count 1 neuter) failure failure)
(decision (noun count 1 neuter) decision decision)
(diagnosis (noun count 1 neuter) diagnosis decision)
(evaluation (noun count 1 neuter) evaluation evaluation)
(family (noun count 1 neuter) family family)
(patient (noun count 1 neuter) patient patient)
(doctor  (noun count 1 neuter) doctor doctor)
(boy       (noun count 1 masculine) boy boy)
(girl      (noun count 1 feminine) girl girl)
(human     (noun count 1 neuter) human human)
(house     (noun count 1 neuter) house house)

; disorders
(global (noun mass 1 neuter) global global)
(focal (noun mass 1 neuter) focal focal)
(amnesic (noun mass 1 neuter) amnesic amnesic)
(multi-infarct-dementia (noun mass 1 neuter) multi-infarct-dementia multi-infarct-dementia)
(encephalitis (noun mass 1 neuter) encephalitis encephalitis)
(severe-head-trauma (noun mass 1 neuter) severe-head-trauma severe-head-trauma)
(alcohol (noun mass 1 neuter) alcohol alcohol)
(toxicity (noun mass 1 neuter) toxicity toxicity)
(faking (noun mass 1 neuter) faking faking)
(subcortical (noun mass 1 neuter) subcortical subcortical)
(genetics (noun mass 1 neuter) genetics genetics)
(origin (noun mass 1 neuter) origin origin)
(hyper-activity (noun mass 1 neuter) hyper-activity hyper-activity)
(genetic-history (noun mass 1 neuter) genetic-history genetic-history)
(chorea (noun mass 1 neuter) chorea chorea)
(jitters (noun mass 1 neuter) jitters jitters)
(stm-good-iq (noun mass 1 neuter) stm-good-iq memory-iq)
(apathetic (noun mass 1 neuter) apathetic apathetic)
(disinterest (noun mass 1 neuter) disinterest disinterest)
```

```
(name      (noun count 1 neuter) name name)
(class     (noun count 1 neuter) class class)
(sub-class (noun count 1 neuter) sub-class sub-class)
(type      (noun count 1 neuter) type type)
(dda       (noun count 1 neuter) dda dda)

(entity      (noun count 1 neuter) entity entity)
(value       (noun count 1 neuter) value value)
(likelihood (noun count sing3p neuter) likelihood likelihood)
(result      (noun count sing3p neuter) result result)
(damage      (noun mass sing3p neuter) damage damage)
(importance (noun count 1 neuter) importance importance)
(symptom     (noun count plur neuter) symptom symptom)
(test        (noun count plur neuter) test test)
(observation (noun count plur neuter) observation observation)
(observation-class (noun count plur neuter) observation-class observation-class)
(disorder-class (noun count plur neuter) disorder-class disorder-class)
(state     (noun count sing3p neuter) state state)
(category    (noun count sing3p neuter) category category)
(symptom     (noun count sing3p neuter) symptom symptom)
(test        (noun count sing3p neuter) test test)
(observation (noun count sing3p neuter) observation observation)
(observation-class (noun count sing3p neuter) observation-class observation-class)
(disorder-class (noun count sing3p neuter) disorder-class disorder-class)

;-------------------------------------------------------------------------
;                              ADJECTIVES                                 |
;-------------------------------------------------------------------------

(similar (adjective attributive) similar similar)
(different (adjective attributive) different different)
(relative (adjective attributive) relative relative)
(left (adjective attributive) left left)
(right (adjective attributive) right right)
(cognitive (adjective attributive) cognitive cognitive)
(pervasive (adjective attributive) pervasive pervasive)
(local (adjective attributive) local local)
(genetic (adjective attributive) genetic genetic)
(lateral (adjective attributive) lateral lateral)
(quick (adjective attributive) quick quick)
(personal (adjective attributive) personal personal)
(uncontrollable (adjective attributive) uncontrollable uncontrollable)

(damaged (adjective attributive) damaged damaged)
(slow (adjective attributive) slow slow)
(fast (adjective attributive) fast fast)
(drunk (adjective attributive) drunk drunk)

;-------------------------------------------------------------------------
;                              DETERMINERS                                |
;-------------------------------------------------------------------------
; ** articles
(a     (determiner count sing3p indefart notof noneg nonum) (article before consonant) a)
(an    (determiner count sing3p indefart notof noneg nonum) (article before vowel) an)
(the   (determiner count 1 defart notof noneg nonum) (sing/plur form of the) the)

;-------------------------------------------------------------------------
;                              PREPOSITIONS                               |
;-------------------------------------------------------------------------

(to (preposition) (toward or in the direction of) to)
(in (preposition) (inner or inward location) in)
(with (preposition) (connection or association) with)
(from (preposition) (place of origin) from)
(of (preposition) (place of origin) of)
(for (preposition) (indicating purpose) for)

; conjoined prepositions
(located (preposition located-in) (located-in) located)
(in (preposition located-in) (located-in) in)

; verbal particles
(in (preposition en) (contained-in) in)
(by (preposition en) (indicated-by) by)

;-------------------------------------------------------------------------
;                              PRONOUNS                                   |
;-------------------------------------------------------------------------

(he   (pronoun pers sing3p subj p3 masculine) (male) he)
(she  (pronoun pers sing3p subj p3 feminine) (female) she)
(it   (pronoun pers sing3p subj p3 neuter) (a thing) it)
(they (pronoun pers plur   subj p3 neuter) (a group of others) they)

(him  (pronoun pers sing3p obj p3) (a male viewed objectively) him)
```

```
    (her  (pronoun pers sing3p obj p3) (a female viewed objectively) her)
    (it   (pronoun pers sing3p obj p3) (a thing viewed objectively) it)
    (them (pronoun pers plur   obj p3) (a group of others) them)

    (his  (pronoun poss sing3p obj p3) (belonging to a male viewed objectively) his)
    (her  (pronoun poss sing3p obj p3) (belonging to a female viewed objectively) her)
    (its  (pronoun poss sing3p obj p3) (belonging to a thing viewed objectively) its)
    (their (pronoun poss plur  obj p3) (belonging to a group of others) their)

    (his  (pronoun poss sing3p subj p3) (belonging to a male viewed subjectively) his)
    (hers (pronoun poss sing3p subj p3) (belonging to a female viewed subjectively) hers)
    (its  (pronoun poss sing3p subj p3) (belonging to a thing viewed subjectively) its)
    (theirs (pronoun poss plur subj p3) (belonging to a group of others) theirs)

    ; ** relative pronouns
    (that (pronoun rel) (The ball that is red) that)
    (who (pronoun rel) (The patient who died) who)
    (which (pronoun rel) (The book which burned) which)

   ; ** demonstrative pronouns
    (this  (pronoun demonstr sing3p) (this book) this)
    (that  (pronoun demonstr sing3p) (that book) that)
    (these (pronoun demonstr plur) (these books) these)
    (those (pronoun demonstr plur) (those books) those)


   ;-----------------------------------------------------------------------------
   ;                              PUNCTUATION                                    |
   ;-----------------------------------------------------------------------------

    (comma  (comma) comma comma)
    (period (period) period period)
    (colon  (colon) colon colon)
   (exclamation-point  (exclamation-point) exclamation-point exclamation-point)
   (question-mark  (question-mark) question-mark question-mark)

   ;-----------------------------------------------------------------------------
   ;                              CONJUNCTIONS                                   |
   ;-----------------------------------------------------------------------------

    (and (conjunction coord) (intersection) and)
    (or (conjunction coord) (union) or)
    (but (conjunction coord) (qualification) but)
    (before (conjunction subord) (pre-temporal) before)
    (after (conjunction subord) (post-temporal) after)
    (because (conjunction subord) (causality) because)

                              CONNECTIVES

    ; there pronoun treated as a connective, unsure of syntactic analysis
    (there (pronoun pers plur  subj p3 neuter) (there are) there)
    (for   (connective for-example) for for)
    (example  (connective for-example) example example)
    (instance (connective) instance instance)
    (therefore (connective) therefore therefore)
    (because (connective) because because)
   ))
```

```
;////////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                               |
;  MODULE:     NEUROPSYCHOLOGIST FRAME KNOWLEDGE BASE                           |
;  PURPOSE:    To represent structural knowledge of the domain.                 |
;  OWNER:      copywrite Mark T. Maybury, July, 1987.                           |
;              Developed from Maybury and Weiss, 1986.                          |
;  REFERENCE:  Knowledge Structures from [Minsky, 1975] frame representation    |
;              formalism.                                                       |
;                                                                               |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////////

;-------------------------------------------------------------------------------
;                                                                               |
;  BRAIN_KB  contains a model of the brain which is organized as a hierarchy.   |
;     It is broken down into left and right hemispheres which are further       |
;     subdivided into local lobes.  Each lobe has associated conditions (the    |
;     children frame) which are present when there is damage in that            |
;     particular lobe.  The sibling frames of each of these conditions,         |
;     furthermore, are demons which calculate a probability from |1 to +1.      |
;     These demons are test results or professional observations which          |
;     indicate the presence or absence of the parent condition.                 |
;                                                                               |
;                                                                               |
;-------------------------------------------------------------------------------

(construct-frame-kb '(

(human (super-class (value nil))    ; or organ
       (sub-class   (brain heart lungs disorder))
       (type        (value body)))  ; or region or object

(brain (super-class (value human))
       (sub-class   (value left-hemisphere right-hemisphere))
       (type        (value region))
       (dda         (value (location skull human) (function understanding)))
       (importance  (value 10))
       (damage      (value 5)))

(left-hemisphere
           (super-class (value brain))
           (sub-class   (value lfrontal  lparietal lsubcortex
                               ltemporal loccipital))
           (type        (value region))
           (dda         (value (location brain left) (function feature-recognition)))
           (importance  (value 10))
           (damage      (value 3)))

(right-hemisphere
           (super-class (value brain))
           (sub-class   (value rfrontal  rparietal rsubcortex
                               rtemporal roccipital))
           (type        (value region))
           (dda         (value (location brain right) (function gestalt-understanding)))
           (importance  (value 10))
           (damage      (value 7)))

(lfrontal (super-class (value left-hemisphere))
          (sub-class   (value mental-control l-cog-flexibility
                              r-hemi-paralysis language comprehension
                              negative-mood movement writing))
          (dda         (value (location hemisphere left) (function control)))
          (type        (value lobe))
          (importance  (value 5))
          (damage      (value 4)))

    (l-cog-flexibility (super-class (value lfrontal))
                       (sub-class   (value m-n m-n-perseveration
                                           loops loops-perseveration
                                           wisconsin)
                                    (importance 1 1 1 1 1 1 1))
                       (dda         (value (function control cognitive)))
                       (type        (value symptom))
                       (importance  (value 3))
                       (damage      (value 8)))

        (m-n                    (super-class (value l-cog-flexibility))
                                (type        (value observation))
                                (dda         (value (function drawings m-n)
                                                    (instrument pencil)))
                                (damage      (value 5)))
        (m-n-perseveration      (super-class (value l-cog-flexibility))
                                (type        (value observation))
                                (dda         (value (function scribbles m-n)
                                                    (instrument pencil)))
```

```
                                    (damage      (value 7)))
                (loops              (super-class (value l-cog-flexibility))
                                    (type        (value observation))
                                    (dda         (value (function drawings loop)
                                                        (instrument pencil)))
                                    (damage      (value 9)))
                (loops-perseveration (super-class (value l-cog-flexibility))
                                    (type        (value observation))
                                    (dda         (value (function scribbles loops)
                                                        (instrument pencil)))
                                    (damage      (value 6)))
                (wisconsin          (super-class (value l-cog-flexibility
                                                        r-cog-flexibility))
                                    (type        (value observation))
                                    (dda         (value (function building house)))
                                    (damage      (value 9)))

          (mental-control   (super-class (value lfrontal))
                            (sub-class   (value immediate-recall)
                                         (importance 1))
                            (type        (value symptom))
                            (dda         (value (function stability memory)))
                            (importance  (value 8))
                            (damage      (value 2)))

              (immediate-recall     (super-class (value mental-control))
                                    (type        (value test))
                                    (dda         (value (function evaluation stm)))
                                    (damage      (value 4)))


     (lparietal (super-class  (value left-hemisphere))
                (sub-class    (value gerstmann-syndrome l-constructional-dyspraxia
                                     reading-comp aphasia))
                (type         (value lobe))
                (dda          (value (location hemisphere left) (function motor-response)))
                (importance   (value 3))
                (damage       (value 4)))


     (lsubcortex (super-class (value left-hemisphere))
                 (sub-class   (value short-term-memory right-body-control))
                 (type        (value lobe))
                 (dda         (value (location hemisphere lower nil left) (function sensation))
)
                 (importance  (value 5))
                 (damage      (value 2)))

     (ltemporal (super-class (value left-hemisphere))
                (sub-class   (value naming instability))
                (type        (value lobe))
                (dda         (value (location hemisphere lateral nil nil left) (function langua
ge)))
                (importance  (value 8))
                (damage      (value 3)))

          (naming       (super-class (value ltemporal))
                        (sub-class   (value bos-name)
                                     (importance 1))
                        (type        (value symptom))
                        (dda         (value (function failure memory)))
                        (importance  (value 10))
                        (damage      (value 2)))

               ;defined above in aphasia

          (instability  (super-class (value ltemporal))
                        (sub-class   (value personality sex-activity)
                                     (importance 1 1))
                        (type        (value symptom))
                        (dda         (value (function control personal)))
                        (importance  (value 10))
                        (damage      (value 4)))

               (personality         (super-class (value instability))
                                    (type        (value observation))
                                    (dda         (value (function stability-detection) (locati
on patient)))
                                    (importance  (value 10))
                                    (damage      (value 4)))

               (sex-activity        (super-class (value instability))
                                    (type        (value observation))
                                    (dda         (value (function intrapersonal-behavior)))
                                    (importance  (value 10))
                                    (damage      (value 4)))
```

```
(loccipital (super-class (value left-hemisphere))
            (sub-class   (value right-blindness))
            (type        (value lobe))
            (dda         (value (location skull human) (function vision)))
            (importance  (value 3))
            (damage      (value 2)))

(rfrontal (super-class  (value right-hemisphere))
          (sub-class    (value l-hemi-paralysis r-constructional-dyspraxia
                               positive-mood r-cog-flexibility))
          (type         (value lobe))
          (dda          (value (location skull human) (function comprehension)))
          (importance   (value 10))
          (damage       (value 7)))

(rparietal (super-class (value right-hemisphere))
           (sub-class   (value neglectful))
           (type        (value lobe))
           (dda         (value (location skull human) (function motor-response)))
           (importance  (value 10))
           (damage      (value 7)))

(rsubcortex (super-class (value right-hemisphere))
            (sub-class   (value visual-stm left-body-control))
            (type        (value lobe))
            (dda         (value (location skull human) (function sensation)))
            (importance  (value 10))
            (damage      (value 7)))


(rtemporal  (super-class (value right-hemisphere))
            (sub-class   (value no-facial-recognition))
            (type        (value lobe))
            (dda         (value (location skull human) (function language)))
            (importance  (value 10))
            (damage      (value 7)))

(roccipital (super-class (value right-hemisphere))
            (sub-class   (value left-blindness))
            (type        (value lobe))
            (dda         (value (location skull human) (function vision)))
            (importance  (value 10))
            (damage      (value 7)))

(disorder   (super-class (value brain))
            (sub-class   (value global focal amnesic))
            (type        (value category))
            (dda         (value (function evaluation patient)))
            (importance  (value 10))
            (damage      (value 8)))

(global     (super-class (value disorder))
            (sub-class   (value multi-infarct-dementia encephalitis alzheimers
                                severe-head-trauma alcohol toxicity faking))
            (type        (value disorder-class))
            (dda         (value (function damage pervasive) (location brain)))
            (importance  (value 10))
            (damage      (value 9)))

(focal       (super-class (value disorder))                          ;FOCAL
             (sub-class   (value frontal head-trauma stroke
                                 tumor demyelination))
             (type        (value disorder-class))
             (dda         (value (function damage local) (location lobe)))
             (importance  (value 10))
             (damage      (value 3)))


  (huntingtons (super-class (value frontal))              ;GENETIC (PARENTS)
               (sub-class   (value subcortical genetics hyper-activity))
               (type        (value disorder))
               (dda         (value (instrument origin genetic) (location lobe subcort
))
               (importance  (value 1))
               (damage      (value 3)))

      (genetics    (super-class (value huntingtons))
                   (sub-class   (value genetic-history)
                                (importance 1))
                   (type        (value observation-class))
                   (dda         (value (function history disorder) (location family)))
                   (importance  (value 4))
                   (damage      (value 2)))

          (genetic-history (super-class (value genetics))
```

```
                                        (type        (value observation))
                                        (damage      (value 7)))

            (hyper-activity
                        (super-class (value huntingtons))
                        (sub-class   (value chorea)
                                     (importance 1))
                        (type        (value observation-class))
                        (dda         (value (function movement uncontrollable)))
                        (importance  (value 5))
                        (damage      (value 3)))

                (chorea  (super-class (value hyper-activity))
                         (dda         (value (function dancing uncontrollable)
                                             (instrument jitters)))
                         (type        (value observation))
                         (damage      (value 7)))

        (amnesic    (super-class (value disorder))
                    (sub-class   (value korsakoffs))
                    (type        (value disorder-class))
                    (dda         (value (function damage memory) (location brain)))
                    (importance  (value 10))
                    (damage      (value 8)))

          (korsakoffs (super-class (value stm))
                      (sub-class   (value stm-good-iq apathetic)
                                   (importance 1 1))
                      (type        (value disorder))
                      (dda         (value (instrument memory)))
                      (importance  (value 8))
                      (damage      (value 9)))

            (stm-good-iq
                        (super-class (value korsakoffs))
                        (dda         (value (function memory quick)))
                        (type        (value observation))
                        (importance  (value 4))
                        (damage      (value 9)))
            (apathetic
                        (super-class (value korsakoffs))
                        (dda         (value (function disinterest)))
                        (type        (value observation))
                        (importance  (value 2))
                        (damage      (value 10)))



         ) ; close frame list
        ) ; close make frame kb
```

```
;*****************************************************************************
;                                                                            *
;  BRAIN_KB.LSP contains a model of the brain which is organized as a        *
;     hierarchy.  It                                                         *
;     is broken down into left and right hemispheres which are further sub-  *
;     divided into local lobes.  Each lobe has associated conditions (the    *
;     children frame) which are present when there is damage in that         *
;     particular lobe.  The sibling frames of each of these conditions,      *
;     furthermore, are demons which calculate a probability from -1 to +1.   *
;     These demons are test results or professional observations which       *
;     indicate the presence or absence of the parent condition.              *
;                                                                            *
;*****************************************************************************

(construct-frame-kb '(

(brain (super-class (value organ))
       (sub-class   (value left-hemisphere right-hemisphere))
       (type        (value region))
       (importance  (value 1)))

(left-hemisphere
           (super-class (value brain))
           (sub-class   (value lfrontal  lparietal lsubcortex
                               ltemporal loccipital))
           (type        (value region))
           (importance  (value 1)))

(right-hemisphere
           (super-class (value brain))
           (sub-class   (value rfrontal  rparietal rsubcortex
                               rtemporal roccipital))
           (type        (value region))
           (importance  (value 1)))


(lfrontal (super-class (value left-hemisphere))
          (sub-class   (value r-hemi-paralysis language comprehension
                              negative-mood movement l-cog-flexibility
                              mental-control writing))
          (type        (value region))
          (importance  (value 1)))


     (r-hemi-paralysis (super-class (value lfrontal))
                       (sub-class   (value right-finger face-walk)
                                    (importance 1 1))
                       (type        (value symptom))
                       (importance  (value 1)))

          (right-finger          (super-class (value r-hemi-paralysis))
                                 (type        (value observation)))
          (face-walk             (super-class (value r-hemi-paralysis))
                                 (type        (value observation)))

     (language          (super-class (value lfrontal))
                        (sub-class   (value bos-name-no-cues
                                            bos-name-perseveration
                                            fas-generation fas-perseveration
                                            written-difficulty vocab)
                                     (importance 1 1 1 1 1 1))
                        (type        (value symptom))
                        (importance  (value 1)))

          (bos-name-no-cues     (super-class (value language))
                                (type        (value test)))
          (bos-name-perseveration (super-class (value language))
                                  (type        (value test)))
          (fas-generation       (super-class (value language l-cog-flexibility)
                                (type        (value test)))
          (fas-perseveration    (super-class (value language l-cog-flexibility)
                                (type        (value observation)))
          (written-difficulty   (super-class (value language))
                                (type        (value observation)))
          (vocab                (super-class (value language))
                                (type        (value test)))

     (comprehension     (super-class (value lfrontal))
                        (sub-class   (value verbal-abstract-reasoning
                                            similarities)
                                     (importance 1 1))
                        (type        (value symptom))
                        (importance  (value 1)))

          (verbal-abstract-reasoning (super-class (value comprehension))
                                     (type        (value test)))
```

```
                    (similarities              (super-class (value comprehension))
                                               (type        (value test)))

               (negative-mood     (super-class (value lfrontal))
                                  (sub-class   (value depression)
                                               (importance 1))
                                  (type        (value symptom))
                                  (importance  (value 1)))

                    (depression   (super-class (value negative-mood))
                                  (type        (value observation)))

               (movement          (super-class (value lfrontal))
                                  (sub-class   (value slow)
                                               (importance 1))
                                  (type        (value symptom))
                                  (importance  (value 1)))

                    (slow        (super-class (value movement))
                                 (type        (value observation)))

          (l-cog-flexibility (super-class (value lfrontal))
                             (sub-class   (value m-n m-n-perseveration
                                                 loops loops-perseveration
                                                 fas-generation fas-perseveration
                                                 wisconsin)
                                          (importance 1 1 1 1 1 1 1))
                             (type        (value symptom))
                             (importance  (value 1)))

                    (m-n                  (super-class (value l-cog-flexibility))
                                          (type        (value observation)))
                    (m-n-perseveration    (super-class (value l-cog-flexibility))
                                          (type        (value observation)))
                    (loops                (super-class (value l-cog-flexibility))
                                          (type        (value observation)))
                    (loops-perseveration  (super-class (value l-cog-flexibility))
                                          (type        (value observation)))
                    ;fas-generation defined above in language
                    (fas-perseveration    (super-class (value l-cog-flexibility))
                                          (type        (value observation)))
                    (wisconsin            (super-class (value l-cog-flexibility
                                                              r-cog-flexibility))
                                          (type        (value observation)))

               (mental-control    (super-class (value lfrontal))
                                  (sub-class   (value immediate-recall)
                                               (importance 1))
                                  (type        (value symptom))
                                  (importance  (value 1)))

                    (immediate-recall     (super-class (value mental-control))
                                          (type        (value test)))

               (writing           (super-class (value lfrontal))
                                  (sub-class   (value fluency grammar sequencing
                                                      letter-form)
                                               (importance 1 1 1 1))
                                  (type        (value symptom))
                                  (importance  (value 1)))

                    (fluency              (super-class (value writing))
                                          (type        (value observation)))
                    (grammar              (super-class (value writing aphasia))
                                          (type        (value observation)))
                    (sequencing           (super-class (value writing))
                                          (type        (value observation)))
                    (letter-form          (super-class (value writing))
                                          (type        (value observation)))

          (lparietal (super-class  (value left-hemisphere))
                     (sub-class    (value gerstmann-syndrome l-constructional-dyspraxia
                                          reading-comp aphasia))
                     (type         (value region))
                     (importance   (value 1)))

               (gerstmann-syndrome (super-class (value lparietal))
                                   (sub-class   (value finger-agnosia left-right-confusION
                                                       dyscalculia dysgraphia)
                                                (importance  1 1 1 1))
                                   (type        (value symptom))
                                   (importance  (value 1)))

                    (finger-agnosia       (super-class (value gerstmann-syndrome))
                                          (type        (value test)))
                    (left-right-confusion (super-class (value gerstmann-syndrome))
```

```
                                        (type         (value observation)))
                (dyscalculia            (super-class (value gerstmann-syndrome))
                                        (type         (value observation)))
                (dysgraphia             (super-class (value gerstmann-syndrome))
                                        (type         (value observation)))

        (l-constructional-dyspraxia
                                (super-class (value lparietal))
                                (sub-class   (value drawings blocks block-details)
                                             (importance  1 1 1))
                                (type        (value symptom))
                                (importance  (value 1)))

                (drawings               (super-class (value l-constructional-dyspraxia))
                                        (type         (value observation)))
                (blocks                 (super-class (value l-constructional-dyspraxia))
                                        (type         (value observation)))
                (block-details          (super-class (value l-constructional-dyspraxia))
                                        (type         (value observation)))

        (reading-comp           (super-class (value lparietal))
                                (sub-class   (value dyslexia oral-reading)
                                             (importance 1 1))
                                (type        (value symptom))
                                (importance  (value 1)))

                (dyslexia               (super-class (value reading-comp))
                                        (type         (value observation)))
                (oral-reading           (super-class (value reading-comp))
                                        (type         (value observation)))

        (aphasia                (super-class (value lparietal))
                                (sub-class   (value bos-name bos-name-paraphasia
                                                    neologisms grammar syntax)
                                             (importance 1 1 1 1 1))
                                (type        (value symptom))
                                (importance  (value 1)))

                (bos-name               (super-class (value aphasia naming))
                                        (type         (value test)))
                (bos-name-paraphasia    (super-class (value language))
                                        (type         (value observation)))
                (neologisms             (super-class (value aphasia))
                                        (type         (value observation)))
                ;grammar defined above in writing
                (syntax                 (super-class (value aphasia))
                                        (type         (value observation)))

(lsubcortex (super-class (value left-hemisphere))
            (sub-class   (value short-term-memory right-body-control))
            (type        (value region))
            (importance  (value 1)))

        (short-term-memory
                        (super-class (value lsubcortex))
                        (sub-class   (value digits logical-memory paired-associateS)
                                     (importance 1 1 1))
                        (type        (value symptom))
                        (importance  (value 1)))

                (digits                 (super-class (value short-term-memory))
                                        (type         (value test)))
                (logical-memory         (super-class (value short-term-memory))
                                        (type         (value test)))
                (paired-associates      (super-class (value short-term-memory))
                                        (type         (value test)))

        (right-body-control
                        (super-class (value lsubcortex))
                        (sub-class   (value right-dominant-finger)
                                     (importance 1))
                        (type        (value symptom))
                        (importance  (value 1)))

                (right-dominant-finger (super-class (value right-body-control))
                                       (type        (value observation)))

(ltemporal (super-class (value left-hemisphere))
           (sub-class   (value naming instability))
           (type        (value region))
           (importance  (value 1)))

        (naming         (super-class (value ltemporal))
                        (sub-class   (value bos-name)
                                     (importance 1))
                        (type        (value symptom))
```

```
                                (importance  (value 1)))

                          ;defined above in aphasia

                (instability  (super-class (value ltemporal))
                              (sub-class   (value personality sex-activity)
                                           (importance 1 1))
                              (type        (value symptom))
                              (importance  (value 1)))

                     (personality          (super-class (value instability))
                                           (type        (value observation)))
                     (sex-activity         (super-class (value instability))
                                           (type        (value observation)))

           (loccipital (super-class (value left-hemisphere))
                       (sub-class   (value right-blindness))
                       (type        (value region))
                       (importance  (value 1)))

                (right-blindness (super-class (value loccipital))
                                 (sub-class   (value r-blindness)
                                              (importance 1))
                                 (type        (value symptom))
                                 (importance  (value 1)))

                     (r-blindness             (super-class (value right-blindness))
                                              (type        (value observation)))

           (rfrontal (super-class  (value right-hemisphere))
                     (sub-class    (value l-hemi-paralysis r-constructional-dyspraxia
                                           positive-mood r-cog-flexibility))
                     (type         (value region))
                     (importance   (value 1)))

                (l-hemi-paralysis (super-class (value rfrontal))
                                  (sub-class   (value left-finger)
                                               (importance 1))
                                  (type        (value symptom))
                                  (importance  (value 1)))

                     (left-finger             (super-class (value l-hemi-paralysis))
                                              (type        (value observation)))

                (r-constructional-dyspraxia
                                  (super-class (value rfrontal))
                                  (sub-class   (value exploded separate-blocks
                                                      picture-misarrangement
                                                      puzzle-difficulty pattern-matching)
                                               (importance 1 1 1 1 1))
                                  (type        (value symptom))
                                  (importance  (value 1)))

                    (exploded             (super-class (value r-constructional-dyspraxia))
                                          (type        (value observation)))
                    (separate-blocks      (super-class (value r-constructional-dyspraxia))
                                          (type        (value observation)))
                  (picture-misarrangement (super-class (value r-constructional-dyspraxia))
                                          (type        (value observation)))
                    (puzzle-difficulty    (super-class (value r-constructional-dyspraxia))
                                          (type        (value observation)))
                    (pattern-matching     (super-class (value r-constructional-dyspraxia))
                                          (type        (value observation)))

                (positive-mood    (super-class (value rfrontal))
                                  (sub-class   (value elation)
                                               (importance 1))
                                  (type        (value symptom))
                                  (importance  (value 1)))

                     (elation             (super-class (value positive-mood))
                                          (type        (value observation)))

               (r-cog-flexibility (super-class (value rfrontal))
                                  (sub-class   (value wisconsin)
                                               (importance 1))
                                  (type        (value symptom))
                                  (importance  (value 1)))

                     ;defined above in l-cog-flexibility

           (rparietal (super-class (value right-hemisphere))
                      (sub-class   (value neglectful))
                      (type        (value region))
                      (importance  (value 1)))
```

```
        (neglectful (super-class (value rparietal))
                    (sub-class   (value neglects-left-space)
                                 (importance 1))
                    (type        (value symptom))
                    (importance  (value 1)))


            (neglects-left-space (super-class (value neglectful))
                                 (type        (value observation)))

    (rsubcortex (super-class (value right-hemisphere))
                (sub-class   (value visual-stm left-body-control))
                (type        (value region))
                (importance  (value 1)))

        (visual-stm  (super-class (value rsubcortex))
                     (sub-class   (value wms-delay visual-reproduction benton)
                                  (importance 1 1 1))
                     (type        (value symptom))
                     (importance  (value 1)))

            (wms-delay           (super-class (value visual-stm))
                                 (type        (value test)))
            (visual-reproduction (super-class (value visual-stm))
                                 (type        (value test)))
            (benton              (super-class (value visual-stm))
                                 (type        (value test)))

        (left-body-control
                      (super-class (value rsubcortex))
                      (sub-class   (value left-dominant-finger)
                                   (importance 1))
                      (type        (value symptom))
                      (importance  (value 1)))

            (left-dominant-finger (super-class (value left-body-control))
                                  (type        (value observation)))

    (rtemporal  (super-class (value right-hemisphere))
                (sub-class   (value no-facial-recognition))
                (type        (value region))
                (importance  (value 1)))

        (no-facial-recognition
                          (super-class (value rtemporal))
                          (sub-class   (value milner-facial-recognition)
                                       (importance 1))
                          (type        (value symptom))
                          (importance  (value 1)))

            (milner-facial-recognition
                                  (super-class (value no-facial-recognition))
                                  (type        (value test)))

    (roccipital (super-class (value right-hemisphere))
                (sub-class   (value left-blindness))
                (type        (value region))
                (importance  (value 1)))

        (left-blindness   (super-class (value roccipital))
                          (sub-class   (value l-blindness)
                                       (importance 1))
                          (type        (value symptom))
                          (importance  (value 1)))

            (l-blindness         (super-class (value left-blindness))
                                 (type        (value observation)))

    )   ;close argument list
    )   ;close CONSTRUCT-FRAME-KB
```

```
;*****************************************************************************
;                                                                            *
;  DISORDER_KB.LSP is the knowledge base which contains a hierarchial        *
;     organisation of organic brain disorders.  Disorders are broken down into *
;     general types (i.e. Global, Focal, or Amnesic) which are               *
;     further sub-divided into specific disorders, each of which is tested for *
;     probability by checking the associated tests or observations which are  *
;     children of the particular disorder frame.                             *
;                                                                            *
;*****************************************************************************

(CONSTRUCT-FRAME-KB '(

(DISORDER    (SUPER-CLASS (VALUE NIL))
             (SUB-CLASS   (VALUE GLOBAL FOCAL AMNESIC))
             (TYPE        (VALUE META-FRAME))
             (IMPORTANCE  (VALUE 1)))

(GLOBAL      (SUPER-CLASS (VALUE DISORDER))
             (SUB-CLASS   (VALUE MULTI-INFARCT-DEMENTIA ENCEPHALITIS ALZHEIMERS
                                 SEVERE-HEAD-TRAUMA ALCOHOL TOXICITY FAKING))
             (TYPE        (VALUE DISORDER-CLASS))
             (IMPORTANCE  (VALUE 1)))


  (MULTI-INFARCT-DEMENTIA
               (SUPER-CLASS (VALUE GLOBAL))
               (SUB-CLASS   (VALUE INSTANT)
                            (IMPORTANCE 1))
               (TYPE        (VALUE DISORDER))
               (IMPORTANCE  (VALUE 1)))

  (ENCEPHALITIS(SUPER-CLASS (VALUE GLOBAL))
               (SUB-CLASS   (VALUE DAYS)
                            (IMPORTANCE 1))
               (TYPE        (VALUE DISORDER))
               (IMPORTANCE  (VALUE 1)))

  (ALZHEIMERS
               (SUPER-CLASS (VALUE GLOBAL))
               (SUB-CLASS   (VALUE MONTHS-YEARS SEVERE-LTM-DAMAGE)
                            (IMPORTANCE 1 1))
               (TYPE        (VALUE DISORDER))
               (IMPORTANCE  (VALUE 1)))

      (SEVERE-LTM-DAMAGE
                   (SUPER-CLASS (VALUE ALZHEIMERS SEVERE-HEAD-TRAUMA ALCOHOL))
                   (TYPE        (VALUE OBSERVATION)))

  (SEVERE-HEAD-TRAUMA
               (SUPER-CLASS (VALUE GLOBAL))
               (SUB-CLASS   (VALUE INSTANT SEVERE-LTM-DAMAGE ACCIDENT)
                            (IMPORTANCE 1 1 1))
               (TYPE        (VALUE DISORDER))
               (IMPORTANCE  (VALUE 1)))

      (ACCIDENT
                   (SUPER-CLASS (VALUE SEVERE-HEAD-TRAUMA HEAD-TRAUMA))
                   (TYPE        (VALUE OBSERVATION)))

  (ALCOHOL
               (SUPER-CLASS (VALUE GLOBAL))   ;IS THIS SEVERE-LTM OR NOT???
               (SUB-CLASS   (VALUE MONTHS-YEARS SEVERE-LTM-DAMAGE ALCOHOLIC)
                            (IMPORTANCE 1 1 1))
               (TYPE        (VALUE DISORDER))
               (IMPORTANCE  (VALUE 1)))

      (ALCOHOLIC
                   (SUPER-CLASS (VALUE ALCOHOL))
                   (TYPE        (VALUE OBSERVATION)))

  (TOXICITY    (SUPER-CLASS (VALUE GLOBAL))
               (SUB-CLASS   (VALUE CHEMICAL-EXPOSURE)
                            (IMPORTANCE 1))
               (TYPE        (VALUE DISORDER))
               (IMPORTANCE  (VALUE 1)))

      (CHEMICAL-EXPOSURE
                   (SUPER-CLASS (VALUE TOXICITY))
                   (TYPE        (VALUE OBSERVATION)))

;IS FAKING ASSOCIATED WITH ONLY GLOBAL DIFFICULTY?
  (FAKING      (SUPER-CLASS (VALUE GLOBAL))
               (SUB-CLASS   (VALUE FAKING-IT)
                            (IMPORTANCE 1))
               (TYPE        (VALUE DISORDER))
```

```
                    (IMPORTANCE  (VALUE 1)))

          (FAKING-IT
                    (SUPER-CLASS (VALUE FAKING))
                    (TYPE        (VALUE OBSERVATION)))

        ;IS ALCOHOL A REFINEMENT OF TOXICITY OR IS IT STRICTLY GLOBAL DISORDER

          (INSTANT (SUPER-CLASS (VALUE MULTI-INFARCT-DEMENTIA SEVERE-HEAD-TRAUMA
                                       TOXICITY HEAD-TRAUMA STROKE ))
                   (TYPE        (VALUE OBSERVATION)))
          (DAYS    (SUPER-CLASS (VALUE ENCEPHALITIS PICKS))
                   (TYPE        (VALUE OBSERVATION)))
          (MONTHS-YEARS
                   (SUPER-CLASS (VALUE ALZHEIMERS NORMAL-PRESSURE-HYDROCEPHALUS
                                       SUPRANUCLEAR-PALSY ALCOHOL))
                   (TYPE        (VALUE OBSERVATION)))

        (FOCAL       (SUPER-CLASS (VALUE DISORDER))                          ;focal
                     (SUB-CLASS   (VALUE FRONTAL HEAD-TRAUMA STROKE
                                         TUMOR DEMYELINATION))
                     (TYPE        (VALUE DISORDER-CLASS))
                     (IMPORTANCE  (VALUE 1)))

          (HEAD-TRAUMA (SUPER-CLASS (VALUE FOCAL))
                       (SUB-CLASS   (VALUE INSTANT MINOR-LTM-DAMAGE ACCIDENT)
                                    (IMPORTANCE  1 1 1))
                       (TYPE        (VALUE DISORDER))
                       (IMPORTANCE  (VALUE 1)))

          (MINOR-LTM-DAMAGE
                       (SUPER-CLASS (VALUE HEAD-TRAUMA))
                       (TYPE        (VALUE OBSERVATION)))

          (STROKE      (SUPER-CLASS (VALUE FOCAL))
                       (SUB-CLASS   (VALUE INSTANT)
                                    (IMPORTANCE 1))
                       (TYPE        (VALUE DISORDER))
                       (IMPORTANCE  (VALUE 1)))
          (TUMOR       (SUPER-CLASS (VALUE FOCAL))
                       (SUB-CLASS   (VALUE TUMOR-EVIDENCE) ;onset could be slow or rapid
                                    (IMPORTANCE 1))
                       (TYPE        (VALUE DISORDER))
                       (IMPORTANCE  (VALUE 1)))

          (TUMOR-EVIDENCE
                       (SUPER-CLASS (VALUE TUMOR))
                       (TYPE        (VALUE OBSERVATION)))
          (DEMYELINATION
                       (SUPER-CLASS (VALUE FOCAL))         ;onset could be slow or rapid
                       (SUB-CLASS   (VALUE DEMYELINATION-EVIDENCE)
                                    (IMPORTANCE 1))
                       (TYPE        (VALUE DISORDER))
                       (IMPORTANCE  (VALUE 1)))

          (DEMYELINATION-EVIDENCE
                       (SUPER-CLASS (VALUE DEMYELINATION))
                       (TYPE        (VALUE OBSERVATION)))

        (FRONTAL     (SUPER-CLASS (VALUE FOCAL))
                     (SUB-CLASS   (VALUE PICKS PARKINSONS HUNTINGTONS
                                         NORMAL-PRESSURE-HYDROCEPHALUS SUPRANUCLEAR-PALSY))
                     (TYPE        (VALUE DISORDER-CLASS))
                     (IMPORTANCE  (VALUE 1)))

          (PICKS       (SUPER-CLASS (VALUE FRONTAL))
                       (SUB-CLASS   (VALUE DAYS BI-FRONTAL)          ;bi-frontal
                                    (IMPORTANCE 1 1))
                       (TYPE        (VALUE DISORDER))
                       (IMPORTANCE  (VALUE 1)))

             (BI-FRONTAL (SUPER-CLASS (VALUE PICKS))
                         (SUB-CLASS   (VALUE LEFT-FRONTAL RIGHT-FRONTAL)
                                      (IMPORTANCE 1 1))
                         (TYPE        (VALUE OBSERVATION-CLASS))
                         (IMPORTANCE  (VALUE 1)))

          (LEFT-FRONTAL
                       (SUPER-CLASS (VALUE BI-FRONTAL))
                       (TYPE        (VALUE OBSERVATION)))
          (RIGHT-FRONTAL
                       (SUPER-CLASS (VALUE BI-FRONTAL))
                       (TYPE        (VALUE OBSERVATION)))


        ;FRONTAL-SUB-CORTICAL with movement disorder to differentiate
```

```
(PARKINSONS  (SUPER-CLASS (VALUE FRONTAL))                ;tremor
             (SUB-CLASS   (VALUE SUBCORTICAL TREMOR)
                          (IMPORTANCE 1 1))
             (TYPE        (VALUE DISORDER))
             (IMPORTANCE  (VALUE 1)))

    (SUBCORTICAL (SUPER-CLASS (VALUE PARKINSONS HUNTINGTONS
                                     NORMAL-PRESSURE-HYDROCEPHALUS
                                     SUPRANUCLEAR-PALSY))
                 (SUB-CLASS   (VALUE LEFT-SUBCORTEX RIGHT-SUBCORTEX)
                              (IMPORTANCE 1 1))
                 (TYPE        (VALUE OBSERVATION-CLASS))
                 (IMPORTANCE  (VALUE 1)))
       (LEFT-SUBCORTEX
                 (SUPER-CLASS (VALUE SUBCORTICAL))
                 (TYPE        (VALUE OBSERVATION)))
       (RIGHT-SUBCORTEX
                 (SUPER-CLASS (VALUE SUBCORTICAL))
                 (TYPE        (VALUE OBSERVATION)))

    (TREMOR
                (SUPER-CLASS (VALUE PARKINSONS))
                (TYPE        (VALUE OBSERVATION)))

(HUNTINGTONS (SUPER-CLASS (VALUE FRONTAL))                ;genetic (parents)
             (SUB-CLASS   (VALUE SUBCORTICAL GENETICS HYPER-ACTIVITY))
             (TYPE        (VALUE DISORDER))
             (IMPORTANCE  (VALUE 1)))

    (GENETICS    (SUPER-CLASS (VALUE HUNTINGTONS))
                 (SUB-CLASS   (VALUE GENETIC-HISTORY)
                              (IMPORTANCE 1))
                 (TYPE        (VALUE OBSERVATION-CLASS))
                 (IMPORTANCE  (VALUE 1)))

        (GENETIC-HISTORY (SUPER-CLASS (VALUE GENETICS))
                         (TYPE        (VALUE OBSERVATION)))

    (HYPER-ACTIVITY
                 (SUPER-CLASS (VALUE HUNTINGTONS))
                 (SUB-CLASS   (VALUE CHOREA)              ;dancing
                              (IMPORTANCE 1))
                 (TYPE        (VALUE OBSERVATION-CLASS))
                 (IMPORTANCE  (VALUE 1)))

        (CHOREA  (SUPER-CLASS (VALUE HYPER-ACTIVITY))
                 (TYPE        (VALUE OBSERVATION)))

(NORMAL-PRESSURE-HYDROCEPHALUS
             (SUPER-CLASS (VALUE FRONTAL))
             (SUB-CLASS   (VALUE SUBCORTICAL OLDER-THAN-60 MONTHS-YEARS)
                          (IMPORTANCE 1 1 1))
             (TYPE        (VALUE DISORDER))
             (IMPORTANCE  (VALUE 1)))

    (OLDER-THAN-60
                (SUPER-CLASS (VALUE NORMAL-PRESSURE-HYDROCEPHALUS))
                (TYPE        (VALUE OBSERVATION)))

(SUPRANUCLEAR-PALSY
             (SUPER-CLASS (VALUE FRONTAL))
             (SUB-CLASS   (VALUE SUBCORTICAL MONTHS-YEARS
                                 CANNOT-MOVE-EYES-UPWARD
                                 BLANK-FACIAL-EXPRESSION)
                          (IMPORTANCE 1 1 1 1))
             (TYPE        (VALUE DISORDER))
             (IMPORTANCE  (VALUE 1)))

    (CANNOT-MOVE-EYES-UPWARD
                (SUPER-CLASS (VALUE SUPRANUCLEAR-PALSY))
                (TYPE        (VALUE OBSERVATION)))
    (BLANK-FACIAL-EXPRESSION
                (SUPER-CLASS (VALUE SUPRANUCLEAR-PALSY))
                (TYPE        (VALUE OBSERVATION)))


(AMNESIC     (SUPER-CLASS (VALUE DISORDER))
             (SUB-CLASS   (VALUE KORSAKOFFS))
             (TYPE        (VALUE DISORDER-CLASS))
             (IMPORTANCE  (VALUE 1)))

  (KORSAKOFFS (SUPER-CLASS (VALUE STM))
              (SUB-CLASS   (VALUE STM-GOOD-IQ APATHETIC)
                           (IMPORTANCE 1 1))
              (TYPE        (VALUE DISORDER)))
```

```
                        (IMPORTANCE  (VALUE 1)))

                (STM-GOOD-IQ
                        (SUPER-CLASS (VALUE KORSAKOFFS))
                        (TYPE        (VALUE OBSERVATION)))
                (APATHETIC
                        (SUPER-CLASS (VALUE KORSAKOFFS))
                        (TYPE        (VALUE OBSERVATION)))


        ) ;close parameter list
        ) ;close CONTRUCT-FRAME-KB
```

PHOTOGRAPHY DICTIONARY AND KNOWLEDGE BASE

```
;////////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                               |
;  MODULE:    PHOTOGRAPHY LEXICON                                               |
;  PURPOSE:   To erase any current dictionary and load in a lexicon of          |
;             photographic terminology.                                         |
;                                                                               |
;  OWNER:     copywrite Mark T. Maybury, July, 1987.                            |
;                                                                               |
;  FORMAT:    < token syntax sematics realization >                             |
;                                                                               |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////////

(erase-dictionary)                              ; defined in makedictionary
(mapc 'make-dictionary-entry '(

; FORMAT: < syntax sematics realization >

;-------------------------------------------------------------------------------
;                                NUMBERS                                        |
;-------------------------------------------------------------------------------

 (one   (number sing3p) (lexical representation of number 1) one)
 (two   (number plur) (lexical representation of number 2) two)
 (three (number plur) (lexical representation of number 3) three)
 (four  (number plur) (lexical representation of number 4) four)
 (five  (number plur) (lexical representation of number 5) five)
 (six   (number plur) (lexical representation of number 6) six)
 (seven (number plur) (lexical representation of number 7) seven)
 (eight (number plur) (lexical representation of number 8) eight)
 (nine  (number plur) (lexical representation of number 9) nine)
 (ten   (number plur) (lexical representation of number 10) ten)

;-------------------------------------------------------------------------------
;                            PROPER NOUNS                                       |
;-------------------------------------------------------------------------------

 (mark (proper-noun sing3p masculine) me mark)
 (michelle (proper-noun sing3p feminine) her michelle)

;-------------------------------------------------------------------------------
;                                VERBS                                          |
;-------------------------------------------------------------------------------

    ; ** the verb "to be"

 (be    (verb copula sing   pres p1)
        (L (_P) (L (_WH) (_P (L (_y) (equal _WH _y))))) am)
 (be    (verb copula sing3p pres p3)
        (L (_P) (L (_WH) (_P (L (_y) (equal _WH _y))))) is)
 (be    (verb copula plur pres p3)
        (L (_P) (L (_WH) (_P (L (_y) (equal _WH _y))))) are)

 (have (verb have-v sing pres p1) (to own or posess) have)
 (have (verb have-v plur   pres p1) (to own or posess) have)
 (have  (verb have-v sing3p pres p3) (to own or posess - irregular 3p sing) has)

 (contain (verb trans plur   pres p3) (restricted or otherwise limited) contain)
 (contain (verb trans sing3p pres p3) (restricted or otherwise limited) contain)

 (indicate (verb trans sing3p pres p1) (telling) indicate)
 (indicate (verb trans plur   pres p3) (telling) indicate)

 (function (verb trans sing3p pres p1) (telling) function)
 (function (verb trans plur   pres p3) (telling) function)

;-------------------------------------------------------------------------------
;                                NOUNS                                          |
;-------------------------------------------------------------------------------


 ; Photographic Fault Classification/Equipment

 (format (noun count sing3p neuter) format format)
 (location (noun count sing3p neuter) location location)
 (process (noun count sing3p neuter) process process)
 (art (noun count sing3p neuter) art art)
 (art-form (noun count sing3p neuter) art-form art-form)
 (visual (noun count sing3p neuter) visual visual)
 (images (noun mass sing3p neuter) images images)
 (equipment (noun mass sing3p neuter) equipment equipment)
 (technique (noun mass sing3p neuter) technique technique)
 (style (noun mass sing3p neuter) style style)

 (illumination (noun mass sing3p neuter) illumination illumination)
```

```
(light (noun count sing3p neuter) light light)
(intensity (noun noun sing3p neuter) intensity intensity)
(lighting (noun mass sing3p neuter) lighting lighting)
(tripod (noun count sing3p neuter) tripod tripod)
(camera (noun count sing3p neuter) camera camera)
(legs (noun count plur neuter) legs legs)
(subject (noun count sing3p neuter) subject subject)

(body (noun count sing3p neuter) body camera-body)
 (film-winder (noun count sing3p neuter) film-winder film-winder)
 (shutter (noun count sing3p neuter) shutter shutter)
 (casing (noun count sing3p neuter) casing casing)
(lens (noun count sing3p neuter) lens lens)
 (diaphram (noun count sing3p neuter) diaphram diaphram)
 (optical-lens (noun count sing3p neuter) optical-lens optical-lens)
 (controls (noun count plur neuter) controls controls)
  (aperture (noun count sing3p neuter) aperture aperture)
  (focal-distance (noun count sing3p neuter) focal-distance focal-distance)
(film (noun mass sing3p neuter) film film)
 (asa (noun count sing3p neuter) asa asa)
 (film-color (noun count sing3p neuter) (color or b/w) film-color)
 (film-type (noun count sing3p neuter) (slide or print) film-type)
 (image-type (noun count sing3p neuter) (slide or print) image-type)
(lighting (noun count sing3p neuter) lighting lighting)
 (no-flash (noun mass sing3p neuter) no-flash no-flash)
 (excess-sun (noun mass sing3p neuter) excess-sun excess-sun)
(composition (noun count sing3p neuter) composition composition)
 (no-subject-balance (noun mass sing3p neuter) no-subject-balance no-subject-balance)
 (bad-positioning (noun mass sing3p neuter) bad-positioning bad-camera-positioning)
(operation (noun count sing3p neuter) operation operation)
 (settings (noun mass plur neuter) settings settings)
 (lens-cap (noun mass sing3p neuter) lens-cap lens-cap)
 (film-loading (noun mass sing3p neuter) film-loading film-loading)
(region      (noun count sing3p neuter) region region)
(expression (noun mass sing3p neuter) expression expression)
(personal      (noun mass sing3p neuter) personal personal)
(exposure      (noun mass sing3p neuter) exposure exposure)
(setting      (noun count sing3p neuter) setting setting)

; capabilities/symptoms

(photography (noun mass sing3p neuter) photography photography)
(protection (noun mass sing3p neuter) protection protection)
(light-pictures (noun mass sing3p neuter) light-pictures light-pictures)
(dark-pictures (noun mass sing3p neuter) dark-pictures dark-pictures)
(lack-of-detail (noun mass sing3p neuter) lack-of-detail lack-of-detail)
(blurred-pictures (noun mass sing3p neuter) blurred-pictures blurred-pictures)
(no-film-loaded (noun mass sing3p neuter) no-film-loaded no-film-loaded)
(example (noun mass sing3p neuter) example example)
(medium (noun count sing3p neuter) medium medium)
(object (noun count sing3p neuter) object object)
(physical (noun mass sing3p neuter) physical physical)
(component (noun count sing3p neuter) component component)
(attribute (noun count sing3p neuter) attribute attribute)
(existance (noun mass sing3p neuter) existance existance)
(method (noun count sing3p neuter) method method)
(wind-film (noun mass sing3p neuter) wind-film wind-film)
(support (noun mass sing3p neuter) support support)
(introduce-light (noun mass sing3p neuter) introduce-light light-introduction)
(focusing (noun mass sing3p neuter) focusing focusing)
(manipulation (noun mass sing3p neuter) manipulation manipulation)
(recording (noun mass sing3p neuter) recording recording)
(impression (noun mass sing3p neuter) impression impression)
(control (noun mass sing3p neuter) control control)
(clarity (noun mass sing3p neuter) clarity clarity)
(winding-spool (noun mass sing3p neuter) winding-spool winding-spool)
(angle (noun count sing3p neuter) angle angle)

(black-white (noun mass sing3p neuter) black-white black-white)
(color (noun mass sing3p neuter) color color)
(infra-red (noun mass sing3p neuter) infra-red infra-red)

(function (noun count sing3p neuter) function function)
(instrument (noun count sing3p neuter) instrument instrument)
(disorder (noun count sing3p neuter) disorder fault)
(fault (noun count plur neuter) fault fault)
(fault (noun count sing3p neuter) fault fault)
(failure (noun count plur neuter) failure failure)
(failure (noun count sing3p neuter) failure failure)
(decision (noun count 1 neuter) decision decision)
(diagnosis (noun count sing3p neuter) diagnosis decision)
(human     (noun count 1 neuter) human human)

(name       (noun count 1 neuter) name name)
(class      (noun count 1 neuter) class class)
(sub-class (noun count 1 neuter) sub-class sub-class)
```

```
(type       (noun count 1 neuter) type type)
(dda        (noun count 1 neuter) dda dda)

(damage     (noun mass sing3p neuter) damage damage)
(entity     (noun count sing3p neuter) entity entity)
(importance (noun count sing3p neuter) importance importance)
(observation (noun count 1 neuter) observation observation)
(symptom    (noun count 1 neuter) symptom symptom)
(test       (noun count 1 neuter) test test)
(value      (noun count sing3p neuter) value value)
(likelihood (noun count sing3p neuter) likelihood likelihood)
(result      (noun count sing3p neuter) result result)

;-------------------------------------------------------------------------
;                              ADJECTIVES                                 |
;-------------------------------------------------------------------------

(artificial (noun count 1 neuter) artificial artificial)
(different (adjective attributive) different different)
(natural (noun count 1 neuter) natural natural)
(physical (adjective attributive) physical physical)
(relative (adjective attributive) relative relative)
(similar (adjective attributive) similar similar)
(visual (adjective attributive) visual visual)
(precise (adjective attributive) precise precise)
(alluminum (adjective attributive) alluminum alluminum)

(traditional (adjective attributive) traditional traditional)
(contemporary (adjective attributive) contemporary contemporary)
(nouveau (adjective attributive) nouveau nouveau)
(no (adjective attributive) no no)

(damaged (adjective attributive) damaged damaged)
(slow (adjective attributive) slow slow)
(fast (adjective attributive) fast fast)

;-------------------------------------------------------------------------
;                              DETERMINERS                                |
;-------------------------------------------------------------------------
; ** articles
(a      (determiner count sing3p indefart notof noneg nonum) (article before consonant) a)
(an     (determiner count sing3p indefart notof noneg nonum) (article before vowel) an)
(the    (determiner count 1 defart notof noneg nonum) (sing/plur form of the) the)

;-------------------------------------------------------------------------
;                              PREPOSITIONS                               |
;-------------------------------------------------------------------------

(to (preposition) (toward or in the direction of) to)
(on (preposition) (outer location or on top of) on)
(in (preposition) (inner or inward location) in)
(with (preposition) (connection or association) with)
(from (preposition) (place of origin) from)
(of (preposition) (place of origin) of)
(for (preposition) (indicating purpose) for)

(located (preposition located-in) (located-in) located)
(in (preposition located-in) (located-in) in)

;-------------------------------------------------------------------------
;                              PRONOUNS                                   |
;-------------------------------------------------------------------------

(he   (pronoun pers sing3p subj p3 masculine) (male) he)
(she  (pronoun pers sing3p subj p3 feminine) (female) she)
(it   (pronoun pers sing3p subj p3 neuter) (a thing) it)
(they (pronoun pers plur   subj p3 neuter) (a group of others) they)

(him  (pronoun pers sing3p obj p3) (a male viewed objectively) him)
(her  (pronoun pers sing3p obj p3) (a female viewed objectively) her)
(it   (pronoun pers sing3p obj p3) (a thing viewed objectively) it)
(them (pronoun pers plur   obj p3) (a group of others) them)

(his  (pronoun poss sing3p obj p3) (belonging to a male viewed objectively) his)
(her  (pronoun poss sing3p obj p3) (belonging to a female viewed objectively) her)
(its  (pronoun poss sing3p obj p3) (belonging to a thing viewed objectively) its)
(their (pronoun poss plur  obj p3) (belonging to a group of others) their)

(his  (pronoun poss sing3p subj p3) (belonging to a male viewed subjectively) his)
(hers (pronoun poss sing3p subj p3) (belonging to a female viewed subjectively) hers)
(its  (pronoun poss sing3p subj p3) (belonging to a thing viewed subjectively) its)
(theirs (pronoun poss plur subj p3) (belonging to a group of others) theirs)

; ** relative pronouns
(that (pronoun rel) (The ball that is red) that)
(who (pronoun rel) (The patient who died) who)
```

```
 (which (pronoun rel) (The book which burned) which)

; ** demonstrative pronouns
 (this  (pronoun demonstr sing3p) (this book) this)
 (that  (pronoun demonstr sing3p) (that book) that)
 (these (pronoun demonstr plur) (these books) these)
 (those (pronoun demonstr plur) (those books) those)


;-------------------------------------------------------------------------------
;                              PUNCTUATION                                      |
;-------------------------------------------------------------------------------

 (comma  (comma) comma comma)
 (period (period) period period)
 (colon  (colon) colon colon)
 (exclamation-point  (exclamation-point) exclamation-point exclamation-point)
 (question-mark  (question-mark) question-mark question-mark)

;-------------------------------------------------------------------------------
;                              CONJUNCTIONS                                     |
;-------------------------------------------------------------------------------

 (and (conjunction coord) (intersection) and)
 (or (conjunction coord) (union) or)
 (but (conjunction coord) (qualification) but)
 (before (conjunction subord) (pre-temporal) before)
 (after (conjunction subord) (post-temporal) after)
 (because (conjunction subord) (causality) because)

;             CONNECTIVES

 (for       (connective for-example) for for)
 (example   (connective for-example) example example)
 (instance  (connective) instance instance)
 (therefore (connective) therefore therefore)
 (because   (connective) because because)

))
```

```
;////////////////////////////////////////////////////////////////////////////////
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;                                                                               |
;  MODULE:     PHOTOGRAPHY FRAME KNOWLEDGE BASE                                 |
;  PURPOSE:    To represent knowledge of photography.                           |
;  OWNER:      copywrite Mark T. Maybury, July, 1987.                           |
;              Special thanks to photographic consultant Neil Russel, CUED.     |
;  REFERENCE:  Knowledge Structures from [Minsky, 1975], frame representation   |
;              formalism.                                                       |
;                                                                               |
;\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
;////////////////////////////////////////////////////////////////////////////////

;-------------------------------------------------------------------------------
;                                                                               |
;  PHOTO.KB  contains a model of the photography process brain which is         |
;     organized as a hierarchy of faults.  Fault diagnosis is broken down       |
;     into equipment analysis, technique evaluation and style investigation.    |
;     Each of these areas are subject to several tests or observations.         |
;                                                                               |
;-------------------------------------------------------------------------------

;  KNOWLEDGE RELATIONSHIPS from SLOTS
;    super/sub-class slots -- part/whole
;    type slot -- type/instance
;  (mechanisms for inheritance of properties)


(construct-frame-kb '(

(expression
       (super-class (value nil))
       (sub-class   (value photography painting eating))
       (dda         (value (attribute requires-talent)))
       (type        (value process)))

(photography (super-class (value expression))
             (sub-class   (value equipment technique style))
             (type        (value art-form))
             (dda         (value (function images recording) (external-location film)))
             (importance  (value 10))
             (damage      (value 5)))

(equipment   (super-class (value photography))
             (sub-class   (value camera lighting tripod))
             (type        (value fault))
             (dda         (value (instrument function physical)
                                 (location camera)))
             (importance  (value 3))
             (damage      (value 4)))

(tripod      (super-class (value equipment))
             (sub-class   (value nil))
             (type        (value instrument))
             (dda         (value (instrument legs alluminum)
                                 (function support camera)))
             (importance  (value 3))
             (damage      (value 1)))

(lighting    (super-class (value equipment))
             (sub-class   (value nil))
             (type        (value instrument))
             (dda         (value (function illumination)
                                 (location subject)))
             (importance  (value 5))
             (damage      (value 9)))

(camera      (super-class (value equipment))
             (sub-class   (value body lens film))
             (type        (value instrument))
             (dda         (value (function images recording) (external-location film)))
             (importance  (value 9))
             (damage      (value 4)))

(body        (super-class (value equipment))
             (sub-class   (value film-winder shutter casing))
             (type        (value component))
             (dda         (value (function support)))
             (importance  (value 9))
             (damage      (value 2)))

(film-winder (super-class (value body))
             (sub-class   (value nil))
             (type        (value component))
             (dda         (value (function wind-film) (location body)))
             (importance  (value 9))
```

```
                    (damage      (value 6)))

          (shutter  (super-class (value body))
                    (sub-class   (value light-pictures dark-pictures blurred-pictures))
                    (type        (value component))
                    (dda         (value (function introduce-light)
                                        (location body)))
                    (importance  (value 8))
                    (damage      (value 2)))

          (casing   (super-class (value body))
                    (sub-class   (value light-pictures))
                    (type        (value component))
                    (dda         (value (function protection film)))
                    (importance  (value 1))
                    (damage      (value 5)))

          (lens     (super-class (value equipment))
                    (sub-class   (value diaphram optical-lens controls))
                    (type        (value component))
                    (dda         (value (function focusing) (location body)))
                    (importance  (value 3))
                    (damage      (value 7)))

          (diaphram (super-class (value lens))
                    (sub-class   (value))
                    (type        (value component))
                    (dda         (value (function introduce-light) (location lens)))
                    (importance  (value 1))
                    (damage      (value 3)))

          (optical-lens (super-class (value lens))
                    (sub-class   (value light-pictures dark-pictures blurred-pictures))
                    (type        (value component))
                    (dda         (value (function focusing) (location camera)))
                    (importance  (value 9))
                    (damage      (value 2)))

          (controls (super-class (value lens))
                    (sub-class   (value aperture focal-distance))
                    (type        (value component))
                    (dda         (value (function manipulation) (location lens)))
                    (importance  (value 2))
                    (damage      (value 9)))

          (aperture (super-class (value controls))
                    (sub-class   (value light-pictures dark-pictures))
                    (type        (value component))
                    (dda         (value (function control intensity none nil light)
                                        (location lens)))
                    (importance  (value 10))
                    (damage      (value 5)))

          (focal-distance
                    (super-class (value controls))
                    (sub-class   (value nil))
                    (type        (value component))
                    (dda         (value (function focusing) (location controls)))
                    (importance  (value 6))
                    (damage      (value 3)))

          (film     (super-class (value equipment))
                    (sub-class   (value image-type film-type asa))
                    (type        (value component))
                    (dda         (value (function recording) (location body)))
                    (importance  (value 4))
                    (damage      (value 4)))

          (image-type (super-class (value film))
                    (sub-class   (value bad-color light-pictures dark-pictures))
                    (type        (value attribute))
                    (dda         (value (function  film)))
                    (importance  (value 8))
                    (damage      (value 6)))

          (film-type (super-class (value film))
                    (sub-class   (value black-white color infra-red))
                    (type        (value attribute))
                    (dda         (value (function ) (location )))
                    (importance  (value 2))
                    (damage      (value 7)))

          (asa      (super-class (value film))
                    (sub-class   (value light-pictures dark-pictures))
                    (type        (value attribute))
                    (dda         (value (function setting exposure)))
```

```
                        (importance  (value 3))
                        (damage      (value 2)))

        (technique      (super-class (value photography))
                        (sub-class   (value lighting composition operation))
                        (type        (value fault))
                        (dda         (value (instrument method precise)))
                        (importance  (value 4))
                        (damage      (value 6)))

        (lighting       (super-class (value technique))
                        (sub-class   (value natural artificial))
                        (type        (value fault))
                        (dda         (value (function impression) (instrument technique)))
                        (importance  (value 10))
                        (damage      (value 8)))

        (composition    (super-class (value technique))
                        (sub-class   (value balance position))
                        (type        (value fault))
                        (dda         (value (function impression) (instrument technique)))
                        (importance  (value 1))
                        (damage      (value 3)))

        (balance        (super-class (value composition))
                        (sub-class   (value inbalance))
                        (type        (value fault))
                        (dda         (value (instrument technique)))
                        (importance  (value 1))
                        (damage      (value 9)))

        (position       (super-class (value composition))
                        (sub-class   (value (function angle) (instrument inbalance)))
                        (type        (value fault))
                        (dda         (value (instrument technique)))
                        (importance  (value 3))
                        (damage      (value 9)))

        (operation      (super-class (value technique))
                        (sub-class   (value settings movement lens-cap film-loading))
                        (type        (value fault))
                        (dda         (value (function control) (instrument camera)))
                        (importance  (value 10))
                        (damage      (value 5)))

        (settings       (super-class (value operation))
                        (sub-class   (value dark-pictures light-pictures
                                            blurred-pictures lack-of-detail))
                        (type        (value fault))
                        (dda         (value (function control)))
                        (importance  (value 1))
                        (damage      (value 4)))

        (movement       (super-class (value operation))
                        (sub-class   (value blurred-pictures))
                        (type        (value fault))
                        (dda         (value (function clarity)))
                        (importance  (value 2))
                        (damage      (value 7)))

        (lens-cap       (super-class (value operation))
                        (sub-class   (value dark-pictures))
                        (type        (value fault))
                        (dda         (value (function protection) (location lens)))
                        (importance  (value 9))
                        (damage      (value 5)))

        (film-loading   (super-class (value operation))
                        (sub-class   (value no-film))
                        (type        (value fault))
                        (dda         (value (function winding-spool) (location camera)))
                        (importance  (value 2))
                        (damage      (value 3)))

        (style          (super-class (value photography))
                        (sub-class   (value traditional contemporary nouveau))
                        (type        (value fault))
                        (dda         (value (instrument expression personal)))
                        (importance  (value 9))
                        (damage      (value 2)))

        (traditional    (super-class (value style))
                        (sub-class   (value British American German French))
                        (type        (value format))  ; style
                        (dda         (value (function expression old-fashioned)))
                        (importance  (value 1))
```

```
                    (damage      (value 5)))

                ;; TESTS AND OBSERVATIONS ;;

        (light-pictures
                    (super-class (value film-type asa aperture shutter casing
                                        optical-lens settings))
                    (type        (value observation))
                    (dda         (value (instrument excess-light)))
                    (importance  (value 5))
                    (damage      (value 6)))

        (dark-pictures
                    (super-class (value lens-cap film-type asa
                                        aperture shutter optical-lens settings))
                    (type        (value observation))
                    (dda         (value (instrument lack-of-light)))
                    (importance  (value 4))
                    (damage      (value 8)))

        (inbalance  (super-class (value balance position))
                    (type        (value observation))
                    (dda         (value (instrument balance no)))
                    (importance  (value 5))
                    (damage      (value 3)))

        (lack-of-detail
                    (super-class (value settings))
                    (type        (value observation))
                    (dda         (value (instrument detail no)))
                    (importance  (value 9))
                    (damage      (value 5)))

        (blurred-pictures
                    (super-class (value shutter optical-lens settings movement))
                    (type        (value observation))
                    (dda         (value (instrument clarity no)))
                    (importance  (value 10))
                    (damage      (value 1)))

        (no-film    (super-class (value film-loading))
                    (type        (value observation))
                    (dda         (value (instrument film no)))
                    (importance  (value 7))
                    (damage      (value 3)))

        (bad-color  (super-class (value film-color))
                    (type        (value observation))
                    (dda         (value (instrument color no)))
                    (importance  (value 10))
                    (damage      (value 5)))

         ) ; close frame list
        ) ; close make frame kb
```

SECTION 13

SYSTEM OUTPUT

```
1       what_is_a_brain?_early.run      Tue Sep  1 11:55:00 1987

   Franz Lisp, Opus 38.79
   -> [load main.l]
   [load support.l]
   [load /user/mphil/mtm/lisp/lispaids/macros.l]
   [load /user/mphil/mtm/lisp/lispaids/io.l]
   [load /user/mphil/mtm/lisp/lispaids/stack.l]
   [load /user/mphil/mtm/lisp/semantics/save.l]
   [load /user/mphil/mtm/lisp/lispaids/track.l]
   [load focus.l]
   [load anaphora.l]
   [load kb_interface.l]
   [load /user/mphil/mtm/dissert/KB/frames.lsp]
   [load /user/mphil/mtm/dissert/KB/construct_kb.lsp]
   [load /user/mphil/mtm/dissert/KB/frame_access.l]
   [load predicates.l]
   [load text.l]
   [load translate.l]
   [load relationalgram.l]
   [fasl generate.o]
   [load realization.l]
   [load morphsyn.l]
   [load surface_form.l]
   [load dictionary.l]
   [load /user/mphil/mtm/lisp/dictionary/dictionary_macros.l]
   [load grammar]
   [load dict]
   [load kb.l]

   -> (main)

   Welcome to the GENNY text generation system for expert systems.
   GENNY was designed to answer questions of the form:

   -- What is an X?
   -- Why did you diagnose Y?  or Why does Y have a problem?
   -- What is the difference between X and Y?

   where X and Y are entities within the provided knowledge base.

   These three types of questions are indicated by the keywords:
   DEFINE, EXPLAIN, and COMPARE, respectively.


   Please enter the domain dictionary file name? neuropsychology.dict
   [load neuropsychology.dict]

   What is the domain of discourse?  neuropsychology.kb
   [load neuropsychology.kb]

   Do you wish DEFINE, EXPLAIN, or COMPARE?  define

   What do you wish to know about?  brain


   TEXT SKETCH:

   introduction
   description
   example

   GENERATE RELEVANT KNOWLEDGE POOL


   GENERATE DISCOURSE SKETCH:
   (definition attributive constituent illustration)

   GLOBAL FOCUS (TOPIC) ==> brain

   LOCAL FOCUS CHOICES (FF/CF/PF) ==> (brain)
   SELECTION ==>
   (definition ((brain))
               ((organ))
               ((location (skull human)) (function (understanding))))

   LOCAL FOCUS CHOICES (FF/CF/PF) ==> (organ brain (brain))
   SELECTION ==>
   (attributive ((brain)) ((value importance indef ten)))

   LOCAL FOCUS CHOICES (FF/CF/PF) ==> (value brain (brain) (brain))
   SELECTION ==>
   (constituent ((brain))
                ((region two none))
                nil
                ((left-hemisphere) (right-hemisphere)))

   LOCAL FOCUS CHOICES (FF/CF/PF) ==> (region left-hemisphere right-hemisphere brain (brain) (brai
```

```
n) (brain))
SELECTION ==>
(illustration ((left-hemisphere))
              ((function feature-recognition))
              ((location (brain))))



= = = = = = = = = = = = = = = = = = = = = = = = =
= = = = = = = = RHETORICAL PREDICATE = = = = = = =
= = = = = = = = = = = = = = = = = = = = = = = = =
(definition ((brain))
            ((organ))
            ((location (skull human)) (function (understanding))))

PRAGMATIC FUNCTION (discourse-topic-entity/focus/given) :

((brain) (nil (brain) (organ)) nil)


SEMANTIC FUNCTION :
action agent patient inst loc funct manner time
(be ((brain)) ((organ)) nil (skull human) (understanding) nil nil nil)


RELATIONAL FUNCTION (voice and form) : (active)

LEXICAL INPUT TO SENTENCE GENERATOR:
((a
  ((determiner count sing3p indefart notof noneg nonum)
   (article before consonant)
   a))
 (brain ((noun count 1 neuter) region brain))
 (be ((copula plur pres p3)
      (L (_P) (L (_WH) (_P (L (_y) (equal _WH _y)))))
      are)
     ((copula sing3p pres p3)
      (L (_P) (L (_WH) (_P (L (_y) (equal _WH _y)))))
      is)
     ((copula sing pres p1)
      (L (_P) (L (_WH) (_P (L (_y) (equal _WH _y)))))
      am))
 (an
  ((determiner count sing3p indefart notof noneg nonum)
   (article before vowel)
   an))
 (organ ((noun count 1 neuter) (cell-based functioning sub component) organ))
 (for ((connective for-example) for for)
      ((preposition) (indicating purpose) for))
 (understanding ((noun mass 1 neuter) consciousness understanding))
 (located ((preposition located-in) (located-in) located))
 (in ((preposition located-in) (located-in) in)
     ((preposition) (inner or inward location) in))
 (the
  ((determiner count 1 defart notof noneg nonum) (sing/plur form of the) the))
 (human ((noun count 1 neuter) human human))
 (skull ((noun count 1 neuter) (cranial container and protector) skull)))

SYNTAX OUTPUT FROM SENTENCE GENERATOR:


(((s declarative active)
  ((np sing3p p3 neuter)
   ((determiner count sing3p indefart notof noneg nonum) ((a)))
   ((n1 sing3p neuter) ((noun count sing3p neuter) ((brain)))))
  ((vp sing3p p3 pres active)
   ((copula sing3p pres p3) ((is)))
   ((np sing3p p3 neuter)
    ((np sing3p p3 neuter)
     ((np sing3p p3 neuter)
      ((determiner count sing3p indefart notof noneg nonum) ((an)))
      ((n1 sing3p neuter) ((noun count sing3p neuter) ((organ)))))
     ((pp)
      ((preposition) ((for)))
      ((np 111 p3 neuter) ((noun mass sing3p neuter) ((understanding))))))
    ((pp)
     ((preposition located-in) ((located)))
     ((preposition located-in) ((in)))
     ((np 33 p3 neuter)
      ((determiner count 21 defart notof noneg nonum) ((the)))
      ((n1 33 neuter)
       ((noun count 27 neuter) ((human)))
       ((noun count 33 neuter) ((skull)))))))))
 ((s declarative active)
  ((np sing3p p3 neuter)
```

```
      ((determiner count sing3p indefart notof noneg nonum) ((a)))
      ((n1 sing3p neuter) ((noun count sing3p neuter) ((brain)))))
     ((vp sing3p p3 pres active)
      ((copula sing3p pres p3) ((is)))
      ((np sing3p p3 neuter)
       ((np sing3p p3 neuter)
        ((determiner count sing3p indefart notof noneg nonum) ((an)))
        ((n1 sing3p neuter) ((noun count sing3p neuter) ((organ)))))
       ((pp)
        ((preposition) ((for)))
        ((np 111 p3 neuter)
         ((np 111 p3 neuter) ((noun mass sing3p neuter) ((understanding))))
         ((pp)
          ((preposition located-in) ((located)))
          ((preposition located-in) ((in)))
          ((np 33 p3 neuter)
           ((determiner count 21 defart notof noneg nonum) ((the)))
           ((n1 33 neuter)
            ((noun count 27 neuter) ((human)))
            ((noun count 33 neuter) ((skull))))))))))))
t



= = = = = = = = = = = = = = = = = = = = = = = = = = = = =
= = = = = = = = RHETORICAL PREDICATE = = = = = = = =
= = = = = = = = = = = = = = = = = = = = = = = = = = = = =
(attributive ((brain)) ((value importance indef ten)))

PRAGMATIC FUNCTION (discourse-topic-entity/focus/given) :

((brain) (((brain)) (brain) (value)) (brain organ))


SEMANTIC FUNCTION :
action agent patient inst loc funct manner time
(have ((brain)) ((value importance indef ten)) nil nil nil nil nil nil)


RELATIONAL FUNCTION (voice and form) : (active)

LEXICAL INPUT TO SENTENCE GENERATOR:
((it ((pronoun pers sing3p subj p3 neuter) (a thing) it))
 (have ((have sing3p pres p3) (to own or posess - irregular |3p| sing) has)
       ((have plur pres p1) (to own or posess) have)
       ((have sing pres p1) (to own or posess) have))
 (an
  ((determiner count sing3p indefart notof noneg nonum)
   (article before vowel)
   an))
 (importance ((noun count 1 neuter) importance importance))
 (value ((noun count 1 neuter) value value))
 (of ((preposition) (place of origin) of))
 (ten ((number plur) (lexical representation of number 10) ten)))

SYNTAX OUTPUT FROM SENTENCE GENERATOR:


(((s declarative active)
  ((np sing3p p3 neuter) ((pronoun pers sing3p subj p3 neuter) ((it))))
  ((vp sing3p p3 pres active)
   ((have sing3p pres p3) ((has)))
   ((np sing3p p3 neuter)
    ((np sing3p p3 neuter)
     ((determiner count sing3p indefart notof noneg nonum) ((an)))
     ((n1 sing3p neuter)
      ((noun count 3 neuter) ((importance)))
      ((noun count sing3p neuter) ((value)))))
    ((pp) ((preposition) ((of))) ((number plur) ((ten))))))))
t



= = = = = = = = = = = = = = = = = = = = = = = = = = = = =
= = = = = = = = RHETORICAL PREDICATE = = = = = = = =
= = = = = = = = = = = = = = = = = = = = = = = = = = = = =
(constituent ((brain))
             ((region two none))
             nil
             ((left-hemisphere) (right-hemisphere)))

PRAGMATIC FUNCTION (discourse-topic-entity/focus/given) :

((brain)
```

```
(((brain) (brain)) (brain) (region left-hemisphere right-hemisphere))
(brain value organ))


SEMANTIC FUNCTION :
action agent patient inst loc funct manner time
(contain ((brain))
         ((region two none))
         nil
         nil
         nil
         ((left-hemisphere) (right-hemisphere))
         nil
         nil)


RELATIONAL FUNCTION (voice and form) : (active colon-insertion)

LEXICAL INPUT TO SENTENCE GENERATOR:
((it ((pronoun pers sing3p subj p3 neuter) (a thing) it))
 (contain ((trans 1 pres 2) (restricted or otherwise limited) contain))
 (two ((number plur) (lexical representation of number 2) two))
 (region ((noun count 1 neuter) region region))
 (colon ((colon) colon colon))
 (the
  ((determiner count 1 defart notof noneg nonum) (sing/plur form of the) the))
 (left-hemisphere ((noun count sing3p neuter) region left-hemisphere))
 (and ((conjunction coord) (intersection) and))
 (the
  ((determiner count 1 defart notof noneg nonum) (sing/plur form of the) the))
 (right-hemisphere ((noun count sing3p neuter) region right-hemisphere)))

SYNTAX OUTPUT FROM SENTENCE GENERATOR:


(((s declarative active)
  ((np sing3p p3 neuter) ((pronoun pers sing3p subj p3 neuter) ((it))))
  ((vp sing3p p3 pres active)
   ((trans sing3p pres 6) ((contain)))
   ((np plur p3 neuter)
    ((np plur p3 neuter)
     ((number plur) ((two)))
     ((n1 plur neuter) ((noun count plur neuter) ((region)))))
    ((colon) ((colon)))
    ((np sing3p p3 neuter)
     ((np sing3p p3 neuter)
      ((determiner count 21 defart notof noneg nonum) ((the)))
      ((n1 sing3p neuter) ((noun count sing3p neuter) ((left-hemisphere)))))
     ((conjunction coord) ((and)))
     ((np sing3p p3 neuter)
      ((determiner count 27 defart notof noneg nonum) ((the)))
      ((n1 sing3p neuter)
       ((noun count sing3p neuter) ((right-hemisphere)))))))))))
 ((s declarative active)
  ((np sing3p p3 neuter) ((pronoun pers sing3p subj p3 neuter) ((it))))
  ((vp sing3p p3 pres active)
   ((trans sing3p pres 6) ((contain)))
   ((np plur p3 neuter)
    ((np plur p3 neuter)
     ((np plur p3 neuter)
      ((number plur) ((two)))
      ((n1 plur neuter) ((noun count plur neuter) ((region)))))
     ((colon) ((colon)))
     ((np sing3p p3 neuter)
      ((determiner count 21 defart notof noneg nonum) ((the)))
      ((n1 sing3p neuter)
       ((noun count sing3p neuter) ((left-hemisphere))))))
    ((conjunction coord) ((and)))
    ((np sing3p p3 neuter)
     ((determiner count 27 defart notof noneg nonum) ((the)))
     ((n1 sing3p neuter)
      ((noun count sing3p neuter) ((right-hemisphere)))))))))
t



= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
= = = = = = = = RHETORICAL PREDICATE = = = = = = = =
= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
(illustration ((left-hemisphere))
              ((function feature-recognition))
              ((location (brain))))

PRAGMATIC FUNCTION (discourse-topic-entity/focus/given) :
```

```
((left-hemisphere)
 (((brain) (brain) (brain)) (left-hemisphere) (feature-recognition))
 (brain region left-hemisphere right-hemisphere value organ))


SEMANTIC FUNCTION :
action agent patient inst loc funct manner time
(have ((left-hemisphere))
      ((function feature-recognition))
      nil
      (brain)
      nil
      nil
      nil
      nil)


RELATIONAL FUNCTION (voice and form) : (active example-insertion)

LEXICAL INPUT TO SENTENCE GENERATOR:
((the
  ((determiner count 1 defart notof noneg nonum) (sing/plur form of the) the))
 (left-hemisphere ((noun count sing3p neuter) region left-hemisphere))
 (comma ((comma) comma comma))
 (for ((connective for-example) for for)
      ((preposition) (indicating purpose) for))
 (example ((connective for-example) example example)
          ((noun mass 1 neuter) example example))
 (comma ((comma) comma comma))
 (have ((have sing3p pres p3) (to own or posess - irregular (3p| sing) has)
        ((have plur pres p1) (to own or posess) have)
        ((have sing pres p1) (to own or posess) have))
 (the
  ((determiner count 1 defart notof noneg nonum) (sing/plur form of the) the))
 (feature-recognition ((noun mass 1 neuter) function feature-recognition))
 (function ((noun count 1 neuter) function function)
           ((verb trans 1 pres 2) (telling) function))
 (located ((preposition located-in) (located-in) located))
 (in ((preposition located-in) (located-in) in)
     ((preposition) (inner or inward location) in))
 (it ((pronoun pers sing3p subj p3 neuter) (a thing) it)))

SYNTAX OUTPUT FROM SENTENCE GENERATOR:


(((s declarative active)
  ((np sing3p p3 neuter)
   ((np sing3p p3 neuter)
    ((determiner count 3 defart notof noneg nonum) ((the)))
    ((n1 sing3p neuter) ((noun count sing3p neuter) ((left-hemisphere)))))
   ((comma) ((comma)))
   ((rel for-example)
    ((connective for-example) ((for)))
    ((connective for-example) ((example))))
   ((comma) ((comma))))
  ((vp sing3p p3 pres active)
   ((have sing3p pres p3) ((has)))
   ((np 27 p3 neuter)
    ((np 27 p3 neuter)
     ((determiner count 15 defart notof noneg nonum) ((the)))
     ((n1 27 neuter)
      ((noun mass 21 neuter) ((feature-recognition)))
      ((noun count 27 neuter) ((function)))))
    ((pp)
     ((preposition located-in) ((located)))
     ((preposition located-in) ((in)))
     ((np sing3p p3 neuter) ((pronoun pers sing3p subj p3 neuter) ((it)))))))))
t

DISCOURSE STRUCTURE + FOCUS + GIVEN

(((definition ((brain))
              ((organ))
              ((location (skull human)) (function (understanding))))
  (nil (brain) (organ))
  nil)
 ((attributive ((brain)) ((value importance indef ten)))
  (((brain)) (brain) (value))
  (brain organ))
 ((constituent ((brain))
               ((region two none))
               nil
               ((left-hemisphere) (right-hemisphere)))
  (((brain) (brain)) (brain) (region left-hemisphere right-hemisphere))
  (brain value organ))
 ((illustration ((left-hemisphere))
```

```
                    ((function feature-recognition))
                    ((location (brain))))
     (((brain) (brain) (brain)) (left-hemisphere) (feature-recognition))
     (brain region left-hemisphere right-hemisphere value organ)))


MESSAGE REALIZATION
((a brain is an organ for understanding located in the human skull)
 (it has an importance value of ten)
 (it contains two regions colon the left-hemisphere and the right-hemisphere)
 (the left-hemisphere
       comma
       for
       example
       comma
       has
       the
       feature-recognition
       function
       located
       in
       it))


SURFACE FORM


A brain is an organ for understanding located in the human skull.
It has an importance value of ten.
It contains two regions: the left-hemisphere and the right-hemisphere.
The left-hemisphere, for example, has the feature-recognition function located in it.
t
-> (exit)
```

```
Franz Lisp, Opus 38.79
-> (include main)
[fasl main.o]

Welcome to the GENNY text generation system for expert systems.
GENNY was designed to answer questions of the form:

-- What is an X?
-- Why did you diagnose Y?  or Why does Y have a problem?
-- What is the difference between X and Y?

where X and Y are entities within the provided knowledge base.

These three types of questions are indicated by the keywords:
DEFINE, EXPLAIN, and COMPARE, respectively.


Please enter the domain dictionary file name? neuropsychology.dict
[load neuropsychology.dict]

What is the domain of discourse?  neuropsychology.kb
[load neuropsychology.kb]

Do you wish DEFINE, EXPLAIN, or COMPARE?  define

What do you wish to know about?  brain


TEXT SKETCH:

introduction
description
example

SELECT KNOWLEDGE VISTA ==> ((brain' brain left-hemisphere right-hemisphere human)

GENERATE RELEVANT PROPOSITION POOL

GENERATE DISCOURSE PLAN:
(definition attributive constituent attributive attributive illustration)

GLOBAL FOCUS (DISCOURSE TOPIC) ==> (brain)

LOCAL FOCUS PREFERENCE ==> (brain)
PREDICATE SELECTED ==>
(definition ((brain))
            ((region))
            ((location (skull human)) (function (understanding))))

LOCAL FOCUS PREFERENCE ==> (region brain)
PREDICATE SELECTED ==>
(attributive ((region brain)) ((value importance indef ten relative)))

LOCAL FOCUS PREFERENCE ==> (brain)
PREDICATE SELECTED ==>
(constituent ((brain))
             ((region two none))
             nil
             ((region left-hemisphere) (region right-hemisphere)))

LOCAL FOCUS PREFERENCE ==> (region left-hemisphere right-hemisphere brain)
PREDICATE SELECTED ==>
(attributive ((region left-hemisphere))
             ((value importance indef ten relative)))

LOCAL FOCUS PREFERENCE ==> (left-hemisphere region right-hemisphere brain)
PREDICATE SELECTED ==>
(attributive ((region right-hemisphere))
             ((value importance indef ten relative)))

LOCAL FOCUS PREFERENCE ==> (right-hemisphere left-hemisphere region brain)
PREDICATE SELECTED ==>
(illustration ((region right-hemisphere))
              ((region))
              ((location (brain right)) (function (gestalt-understanding))))

SURFACE FORM

A brain is a region for understanding located in the human skull.
It has a relative importance value of ten.
It contains two regions: a left-hemisphere region and a right-hemisphere region.
The left-hemisphere region has a relative importance value of ten.
The right-hemisphere region has a relative importance value of ten.
The right-hemisphere region, for example, is a region for gestalt-understanding located in the
right brain.
```

```
PROCESSING TIME
CPU time used for processing: 8826
CPU time used for garbage Collection: 3057

nil
-> (exit)
```

```
gemmy2.out      Tue Sep  1 11:28:49 1987

   Franz Lisp, Opus 38.79
   -> (include main)
   [fasl main.o]
   t

   Please enter the domain dictionary file name? neuropsychology.dict
   [load neuropsychology.dict]

   What is the domain of discourse?  neuropsychology.kb
   [load neuropsychology.kb]

   Do you wish DEFINE, EXPLAIN, or COMPARE? explain

   What do you wish to know about? korsakoffs


   TEXT SKETCH:

   reason
   evidence

   SELECT KNOWLEDGE VISTA ==> ((korsakoffs) korsakoffs stm-good-iq apathetic stm)

   GENERATE RELEVANT PROPOSITION POOL

   GENERATE DISCOURSE PLAN:
   (cause-effect attributive attributive)

   GLOBAL FOCUS (DISCOURSE TOPIC) ==> (korsakoffs)

   LOCAL FOCUS PREFERENCE ==> (korsakoffs)
   PREDICATE SELECTED ==>
   (cause-effect ((disorder korsakoffs))
                 ((manifest))
                 nil
                 ((observation stm-good-iq) (observation apathetic))
                 ((damage)))

   LOCAL FOCUS PREFERENCE ==> (stm-good-iq apathetic manifest damage korsakoffs)
   PREDICATE SELECTED ==>
   (attributive ((observation stm-good-iq)) ((value likelihood indef nine)))

   LOCAL FOCUS PREFERENCE ==> (stm-good-iq apathetic manifest damage korsakoffs)
   PREDICATE SELECTED ==>
   (attributive ((observation apathetic)) ((value likelihood indef ten)))

   SURFACE FORM

   Korsakoffs disorder is manifest because the memory-iq observation and the apathetic
   observation indicate damage.
   The memory-iq observation has a likelihood value of nine.
   The apathetic observation has a likelihood value of ten.

   PROCESSING TIME
   CPU time used for processing: 7632
   CPU time used for garbage Collection: 2718

   nil
   -> (exit)
```

```
Franz Lisp, Opus 38.79
-> (include main)
[fasl main.o]
t

Please enter the domain dictionary file name? neuropsychology.dict
[load neuropsychology.dict]

What is the domain of discourse?  neuropsychology.kb
[load neuropsychology.kb]

Do you wish DEFINE, EXPLAIN, or COMPARE? explain

What do you wish to know about? instability


TEXT SKETCH:

reason
evidence

SELECT KNOWLEDGE VISTA ==> ((instability) instability personality sex-activity ltemporal)

GENERATE RELEVANT PROPOSITION POOL

GENERATE DISCOURSE PLAN:
(cause-effect attributive attributive)

GLOBAL FOCUS (DISCOURSE TOPIC) ==> (instability)

LOCAL FOCUS PREFERENCE ==> (instability)
PREDICATE SELECTED ==>
(cause-effect ((symptom instability))
              ((manifest))
              nil
              ((observation personality) (observation sex-activity))
              ((damage)))

LOCAL FOCUS PREFERENCE ==> (personality sex-activity manifest damage instability)
PREDICATE SELECTED ==>
(attributive ((observation personality)) ((value likelihood indef four)))

LOCAL FOCUS PREFERENCE ==> (personality sex-activity manifest damage instability)
PREDICATE SELECTED ==>
(attributive ((observation sex-activity)) ((value likelihood indef four)))

SURFACE FORM

An instability symptom is manifest because the personality observation and the
sex-activity observation indicate damage.
The personality observation has a likelihood value of four.
The sex-activity observation has a likelihood value of four.

PROCESSING TIME
CPU time used for processing: 7624
CPU time used for garbage Collection: 2694

nil
-> (exit)
```

```
Franz Lisp, Opus 38.79
-> (include main)
[fasl main.o]
t

Please enter the domain dictionary file name? neuropsychology.dict
[load neuropsychology.dict]

What is the domain of discourse?  neuropsychology.kb
[load neuropsychology.kb]

Do you wish DEFINE, EXPLAIN, or COMPARE? explain

What do you wish to know about? personality


TEXT SKETCH:

reason
evidence

SELECT KNOWLEDGE VISTA ==> ((personality) personality instability)

GENERATE RELEVANT PROPOSITION POOL

GENERATE DISCOURSE PLAN:
(cause-effect definition)

GLOBAL FOCUS (DISCOURSE TOPIC) ==> (personality)

LOCAL FOCUS PREFERENCE ==> (personality)
PREDICATE SELECTED ==>
(cause-effect ((observation personality)) ((made)) nil nil ((damage)))

LOCAL FOCUS PREFERENCE ==> (made damage personality)
PREDICATE SELECTED ==>
(definition ((observation personality))
            ((observation))
            ((function (stability-detection)) (location (patient))))


SURFACE FORM

It is an observation for stability-detection located in a patient.

PROCESSING TIME
CPU time used for processing: 7415
CPU time used for garbage Collection: 2683

nil
-> (exit)
```

```
1       gemmy5.out      Tue Sep  1 11:31:53 1987

           Franz Lisp, Opus 38.79
           -> (include main)
           [fasl main.o]
           t

           Please enter the domain dictionary file name? neuropsychology.dict
           [load neuropsychology.dict]

           What is the domain of discourse?  neuropsychology.kb
           [load neuropsychology.kb]

           Do you wish DEFINE, EXPLAIN, or COMPARE? compare

           What do you wish to compare?  personality
           What would you like to compare it to?  sex-activity


           TEXT SKETCH:

           introduction
           introduction
           comparison
           conclusion

           SELECT KNOWLEDGE VISTA ==> ((personality sex-activity) personality sex-activity instability ins
tability)

           GENERATE RELEVANT PROPOSITION POOL

           GENERATE DISCOURSE PLAN:
           (definition attributive definition attributive compare-contrast inference)

           GLOBAL FOCUS (DISCOURSE TOPIC) ==> (personality sex-activity)

           LOCAL FOCUS PREFERENCE ==> (personality sex-activity)
           PREDICATE SELECTED ==>
           (definition ((observation personality))
                       ((observation))
                       ((function (stability-detection)) (location (patient))))

           LOCAL FOCUS PREFERENCE ==> (personality sex-activity observation)
           PREDICATE SELECTED ==>
           (attributive ((observation personality)) ((value likelihood indef four)))

           LOCAL FOCUS PREFERENCE ==> (personality sex-activity)
           PREDICATE SELECTED ==>
           (definition ((observation sex-activity))
                       ((observation))
                       ((function (intrapersonal-behavior))))

           LOCAL FOCUS PREFERENCE ==> (personality sex-activity observation)
           PREDICATE SELECTED ==>
           (attributive ((observation sex-activity)) ((value likelihood indef four)))

           LOCAL FOCUS PREFERENCE ==> (personality sex-activity)
           PREDICATE SELECTED ==>
           (compare-contrast ((personality) (sex-activity))
                             ((class different) (type similar) (importance similar)))

           LOCAL FOCUS PREFERENCE ==> (personality sex-activity class type importance)
           PREDICATE SELECTED ==>
           (inference ((observation personality) (observation sex-activity))
                      ((entity similar none)))

           SURFACE FORM

           A personality observation is an observation for stability-detection located in a patient.
           It has a likelihood value of four.
           A sex-activity observation is an observation for intrapersonal-behavior.
           It has a likelihood value of four.
           Personality and it have a different class, a similar type and a similar importance.
           It and the sex-activity observation, therefore, are similar entities.

           PROCESSING TIME
           CPU time used for processing: 8794
           CPU time used for garbage Collection: 3050

           nil
           -> (exit)
```

```
Franz Lisp, Opus 38.79
-> (include main)
[fasl main.o]
t

Please enter the domain dictionary file name? neuropsychology.dict
[load neuropsychology.dict]

What is the domain of discourse?  neuropsychology.kb
[load neuropsychology.kb]

Do you wish DEFINE, EXPLAIN, or COMPARE? explain

What do you wish to know about? ltemporal


TEXT SKETCH:

reason
evidence

SELECT KNOWLEDGE VISTA ==> ((ltemporal) ltemporal naming instability left-hemisphere)

GENERATE RELEVANT PROPOSITION POOL

GENERATE DISCOURSE PLAN:
(cause-effect attributive attributive)

GLOBAL FOCUS (DISCOURSE TOPIC) ==> (ltemporal)

LOCAL FOCUS PREFERENCE ==> (ltemporal)
PREDICATE SELECTED ==>
(cause-effect ((lobe ltemporal))
              ((damaged nil none))
              nil
              ((symptom naming) (symptom instability))
              ((damage)))

LOCAL FOCUS PREFERENCE ==> (naming instability damaged damage ltemporal)
PREDICATE SELECTED ==>
(attributive ((symptom naming)) ((value likelihood indef two)))

LOCAL FOCUS PREFERENCE ==> (naming instability damaged damage ltemporal)
PREDICATE SELECTED ==>
(attributive ((symptom instability)) ((value likelihood indef four)))

SURFACE FORM

The left-temporal lobe is damaged because the naming symptom
and the instability symptom indicate damage.
The naming symptom has a likelihood value of two.
The instability symptom has a likelihood value of four.


PROCESSING TIME
CPU time used for processing: 7601
CPU time used for garbage Collection: 2699

nil
-> (exit)
```

```
Franz Lisp, Opus 38.79
-> (include main)
[fasl main.o]
t

Please enter the domain dictionary file name? neuropsychology.dict
[load neuropsychology.dict]

What is the domain of discourse?  neuropsychology.kb
[load neuropsychology.kb]

Do you wish DEFINE, EXPLAIN, or COMPARE? define

What do you wish to know about? left-hemisphere


TEXT SKETCH:

introduction
description
example

SELECT KNOWLEDGE VISTA ==> ((left-hemisphere) left-hemisphere lfrontal lparietal lsubcortex lte
mporal loccipital brain)

GENERATE RELEVANT PROPOSITION POOL

GENERATE DISCOURSE PLAN:
(definition attributive
            constituent
            attributive
            attributive
            attributive
            attributive
            attributive
            illustration)

GLOBAL FOCUS (DISCOURSE TOPIC) ==> (left-hemisphere)

LOCAL FOCUS PREFERENCE ==> (left-hemisphere)
PREDICATE SELECTED ==>
(definition ((left-hemisphere))
            ((region))
            ((location (brain left)) (function (feature-recognition))))

LOCAL FOCUS PREFERENCE ==> (region left-hemisphere)
PREDICATE SELECTED ==>
(attributive ((region left-hemisphere))
             ((value importance indef ten relative)))

LOCAL FOCUS PREFERENCE ==> (left-hemisphere)
PREDICATE SELECTED ==>
(constituent ((left-hemisphere))
             ((lobe five none))
             nil
             ((lobe lfrontal)
              (lobe lparietal)
              (lobe lsubcortex)
              (lobe ltemporal)
              (lobe loccipital)))

LOCAL FOCUS PREFERENCE ==> (lobe lfrontal lparietal lsubcortex ltemporal loccipital left-hemisp
here)
PREDICATE SELECTED ==>
(attributive ((lobe lfrontal)) ((value importance indef five relative)))

LOCAL FOCUS PREFERENCE ==> (lfrontal lobe lparietal lsubcortex ltemporal loccipital left-hemisp
here)
PREDICATE SELECTED ==>
(attributive ((lobe lparietal)) ((value importance indef three relative)))

LOCAL FOCUS PREFERENCE ==> (lparietal lfrontal lobe lsubcortex ltemporal loccipital left-hemisp
here)
PREDICATE SELECTED ==>
(attributive ((lobe lsubcortex)) ((value importance indef five relative)))

LOCAL FOCUS PREFERENCE ==> (lsubcortex lparietal lfrontal lobe ltemporal loccipital left-hemisp
here)
PREDICATE SELECTED ==>
(attributive ((lobe ltemporal)) ((value importance indef eight relative)))

LOCAL FOCUS PREFERENCE ==> (ltemporal lsubcortex lparietal lfrontal lobe loccipital left-hemisp
here)
PREDICATE SELECTED ==>
(attributive ((lobe loccipital)) ((value importance indef three relative)))
```

```
LOCAL FOCUS PREFERENCE ==> (loccipital ltemporal lsubcortex lparietal lfrontal lobe left-hemisp
here)
PREDICATE SELECTED ==>
(illustration ((lobe loccipital))
              ((lobe))
              ((location (skull human)) (function (vision))))

SURFACE FORM


A left-hemisphere is a region for feature-recognition located in the left brain.
It has a relative importance value of ten.
It contains five lobes:  the left-frontal lobe, the left-parietal lobe,
the left-subcortex lobe, the left-temporal lobe and the left-occipital lobe.
The left-frontal lobe has a relative importance value of five.
The left-parietal lobe has a relative importance value of three.
The left-subcortex lobe has a relative importance value of five.
The left-temporal lobe has a relative importance value of eight.
The left-occipital lobe has a relative importance value of three.
It, for example, is a lobe for vision located in the human skull.


PROCESSING TIME
CPU time used for processing: 10403
CPU time used for garbage Collection: 3452

nil
-> (exit)
```

```
Franz Lisp, Opus 38.79
-> (include main)
[fasl main.o]
t

Please enter the domain dictionary file name? neuropsychology.dict
[load neuropsychology.dict]

What is the domain of discourse?  neuropsychology.kb
[load neuropsychology.kb]

Do you wish DEFINE, EXPLAIN, or COMPARE? explain

What do you wish to know about? l-cog-flexibility


TEXT SKETCH:

reason
evidence

SELECT KNOWLEDGE VISTA ==> ((l-cog-flexibility) l-cog-flexibility m-n m-n-perseveration loops l
oops-perseveration wisconsin lfrontal)

GENERATE RELEVANT PROPOSITION POOL

GENERATE DISCOURSE PLAN:
(cause-effect attributive attributive attributive attributive attributive)

GLOBAL FOCUS (DISCOURSE TOPIC) ==> (l-cog-flexibility)

LOCAL FOCUS PREFERENCE ==> (l-cog-flexibility)
PREDICATE SELECTED ==>
(cause-effect ((symptom l-cog-flexibility))
              ((manifest))
              nil
              ((observation m-n)
               (observation m-n-perseveration)
               (observation loops)
               (observation loops-perseveration)
               (observation wisconsin))
              ((damage)))

LOCAL FOCUS PREFERENCE ==> (m-n m-n-perseveration loops loops-perseveration wisconsin manifest
damage l-cog-flexibility)
PREDICATE SELECTED ==>
(attributive ((observation m-n)) ((value likelihood indef five)))

LOCAL FOCUS PREFERENCE ==> (m-n m-n-perseveration loops loops-perseveration wisconsin manifest
damage l-cog-flexibility)
PREDICATE SELECTED ==>
(attributive ((observation m-n-perseveration))
             ((value likelihood indef seven)))

LOCAL FOCUS PREFERENCE ==> (m-n-perseveration m-n loops loops-perseveration wisconsin manifest
damage l-cog-flexibility)
PREDICATE SELECTED ==>
(attributive ((observation loops)) ((value likelihood indef nine)))

LOCAL FOCUS PREFERENCE ==> (loops m-n-perseveration m-n loops-perseveration wisconsin manifest
damage l-cog-flexibility)
PREDICATE SELECTED ==>
(attributive ((observation loops-perseveration))
             ((value likelihood indef six)))

LOCAL FOCUS PREFERENCE ==> (loops-perseveration loops m-n-perseveration m-n wisconsin manifest
damage l-cog-flexibility)
PREDICATE SELECTED ==>
(attributive ((observation wisconsin)) ((value likelihood indef nine)))


SURFACE FORM

The left-cognitive-flexibility symptom is manifest because the m-n observation,
the m-n-perseveration observation, the loops observation, the loops-perseveration
the observation and the wisconsin observation indicate damage.
The m-n observation has a likelihood value of five.
The m-n-perseveration observation has a likelihood value of seven.
The loops observation has a likelihood value of nine.
The loops-perseveration observation has a likelihood value of six.
The wisconsin observation has a likelihood value of nine.

PROCESSING TIME
CPU time used for processing: 9001
CPU time used for garbage Collection: 3083
```

```
Franz Lisp, Opus 38.79
-> (include main)
[fasl main.o]
t

Please enter the domain dictionary file name?  photo.dict
[load photo.dict]

What is the domain of discourse?  photo.kb
[load photo.kb]

Do you wish DEFINE, EXPLAIN, or COMPARE?  define

What do you wish to know about?  equipment


TEXT SKETCH:

introduction
description
example

SELECT KNOWLEDGE VISTA ==> ((equipment) equipment camera lighting tripod photography)

GENERATE RELEVANT PROPOSITION POOL

GENERATE DISCOURSE PLAN:
(definition attributive
            constituent
            attributive
            attributive
            attributive
            illustration)

GLOBAL FOCUS (DISCOURSE TOPIC) ==> (equipment)

LOCAL FOCUS PREFERENCE ==> (equipment)
PREDICATE SELECTED ==>
(definition ((fault equipment))
            ((fault))
            ((instrument (function physical)) (location (camera))))

LOCAL FOCUS PREFERENCE ==> (fault equipment)
PREDICATE SELECTED ==>
(attributive ((fault equipment)) ((value importance indef three relative)))

LOCAL FOCUS PREFERENCE ==> (equipment)
PREDICATE SELECTED ==>
(constituent ((equipment))
             ((instrument three none))
             nil
             ((instrument camera) (fault lighting) (instrument tripod)))

LOCAL FOCUS PREFERENCE ==> (instrument camera lighting tripod equipment)
PREDICATE SELECTED ==>
(attributive ((instrument camera)) ((value importance indef nine relative)))

LOCAL FOCUS PREFERENCE ==> (camera instrument lighting tripod equipment)
PREDICATE SELECTED ==>
(attributive ((fault lighting)) ((value importance indef ten relative)))

LOCAL FOCUS PREFERENCE ==> (lighting camera instrument tripod equipment)
PREDICATE SELECTED ==>
(attributive ((instrument tripod)) ((value importance indef three relative)))

LOCAL FOCUS PREFERENCE ==> (tripod lighting camera instrument equipment)
PREDICATE SELECTED ==>
(illustration ((instrument tripod))
              ((instrument))
              ((instrument (legs alluminum)) (function (support camera))))

SURFACE FORM

An equipment fault is a fault with a physical function located in a camera.
It has a relative importance value of three.
It contains three instruments: a camera instrument, a lighting instrument
and a tripod instrument.
The camera instrument has a relative importance value of nine.
The lighting instrument has a relative importance value of ten.
The tripod instrument has a relative importance value of three.
It, for example, is an instrument with alluminum legs for camera support.

PROCESSING TIME
CPU time used for processing: 11147
CPU time used for garbage collection: 3794
```

gemmy10.out Tue Sep 1 11:49:37 1987

```
Franz Lisp, Opus 38.79
-> (include main)
[fasl main.o]
t

Please enter the domain dictionary file name?  photo.dict
[load photo.dict]

What is the domain of discourse?  photo.kb
[load photo.kb]

Do you wish DEFINE, EXPLAIN, or COMPARE?  define

What do you wish to know about?  photography


TEXT SKETCH:

introduction
description
example

SELECT KNOWLEDGE VISTA ==> ((photography) photography equipment technique style expression)

GENERATE RELEVANT PROPOSITION POOL

GENERATE DISCOURSE PLAN:
(definition attributive
            constituent
            attributive
            attributive
            attributive
            illustration)

GLOBAL FOCUS (DISCOURSE TOPIC) ==> (photography)

LOCAL FOCUS PREFERENCE ==> (photography)
PREDICATE SELECTED ==>
(definition ((photography))
            ((art-form))
            ((function (images recording)) (external-location (film))))

LOCAL FOCUS PREFERENCE ==> (art-form photography)
PREDICATE SELECTED ==>
(attributive ((art-form photography)) ((value importance indef ten relative)))

LOCAL FOCUS PREFERENCE ==> (photography)
PREDICATE SELECTED ==>
(constituent ((photography))
             ((fault three none))
             nil
             ((fault equipment) (fault technique) (fault style)))

LOCAL FOCUS PREFERENCE ==> (fault equipment technique style photography)
PREDICATE SELECTED ==>
(attributive ((fault equipment)) ((value importance indef three relative)))

LOCAL FOCUS PREFERENCE ==> (equipment fault technique style photography)
PREDICATE SELECTED ==>
(attributive ((fault technique)) ((value importance indef four relative)))

LOCAL FOCUS PREFERENCE ==> (technique equipment fault style photography)
PREDICATE SELECTED ==>
(attributive ((fault style)) ((value importance indef nine relative)))

LOCAL FOCUS PREFERENCE ==> (style technique equipment fault photography)
PREDICATE SELECTED ==>
(illustration ((fault style)) ((fault)) ((instrument (expression persona))))


SURFACE FORM


Photography is an art-form for recording images on film.
It has a relative importance value of ten.
It contains three faults: an equipment fault, a technique fault and a style fault.
The equipment fault has a relative importance value of three.
The technique fault has a relative importance value of four.
The style fault has a relative importance value of nine.
It, for example, is a fault with personal expression.

PROCESSING TIME
CPU time used for processing: 11143
CPU time used for garbage Collection: 3822

nil
```